Algorithms I

CST Part IA, Easter 2007

12 Lectures

Keir Fraser

`keir.fraser@cl.cam.ac.uk`

Overview

Aims of this course:

- Learn to analyse and reason about an algorithm's performance in terms of time and space complexity.
- Introduce a number of fundamental algorithms relating to sorting and searching of data.
- Discuss a range of techniques for developing new algorithms or extending existing ones.

Reference material:

- Cormen, Leiserson, Rivest and Stein (2001). *Introduction to Algorithms* (2nd ed.). MIT Press.
- Sedgewick (2003). *Algorithms in Java* (3rd ed.). Addison Wesley.

Practice questions

The material in this course is best learnt through textbook *study* and *practice* of exercises and past exam questions.

- Last year's IA Algorithms questions are all relevant.
- Many of the past questions from IB Data Structures & Algorithms are relevant to this course.
- The course materials webpage lists suitable questions back as far as 1997: `http://www.cl.cam.ac.uk/teaching/current/AlgorithI/`
- You can search back even further than this: fundamental algorithms is a stable topic of study!

What is an algorithm?

An algorithm is a precisely defined set of steps that solve a problem. The concept is *fundamental* to computer science:

- An *abstraction* from particular computer systems or programming languages
- Allows *reuse* of solutions across different problems
- Allows *analysis* of algorithm performance

To be considered fundamental, an algorithm will generally solve some very general problem that can then be applied to a wide range of scenarios:

- Search for an integer in a dynamic set
- Sort a list of integers into ascending order
- Find the shortest path between two vertices in a weighted graph

Performance

Many different algorithms can be devised to solve a particular problem. How to choose between them?

- Implementation complexity
- How long it takes to run?
- How much memory does the algorithm require?

Issues of performance and scalability are important.

- The increasing power and storage capacity of computer systems may make some algorithms and larger problems tractable.
- But some problems (or certain *solutions* to those problems) are *fundamentally* intractable.

A simple example

Consider the following *recursive* algorithm for computing the n^{th} value in the Fibonacci sequence:

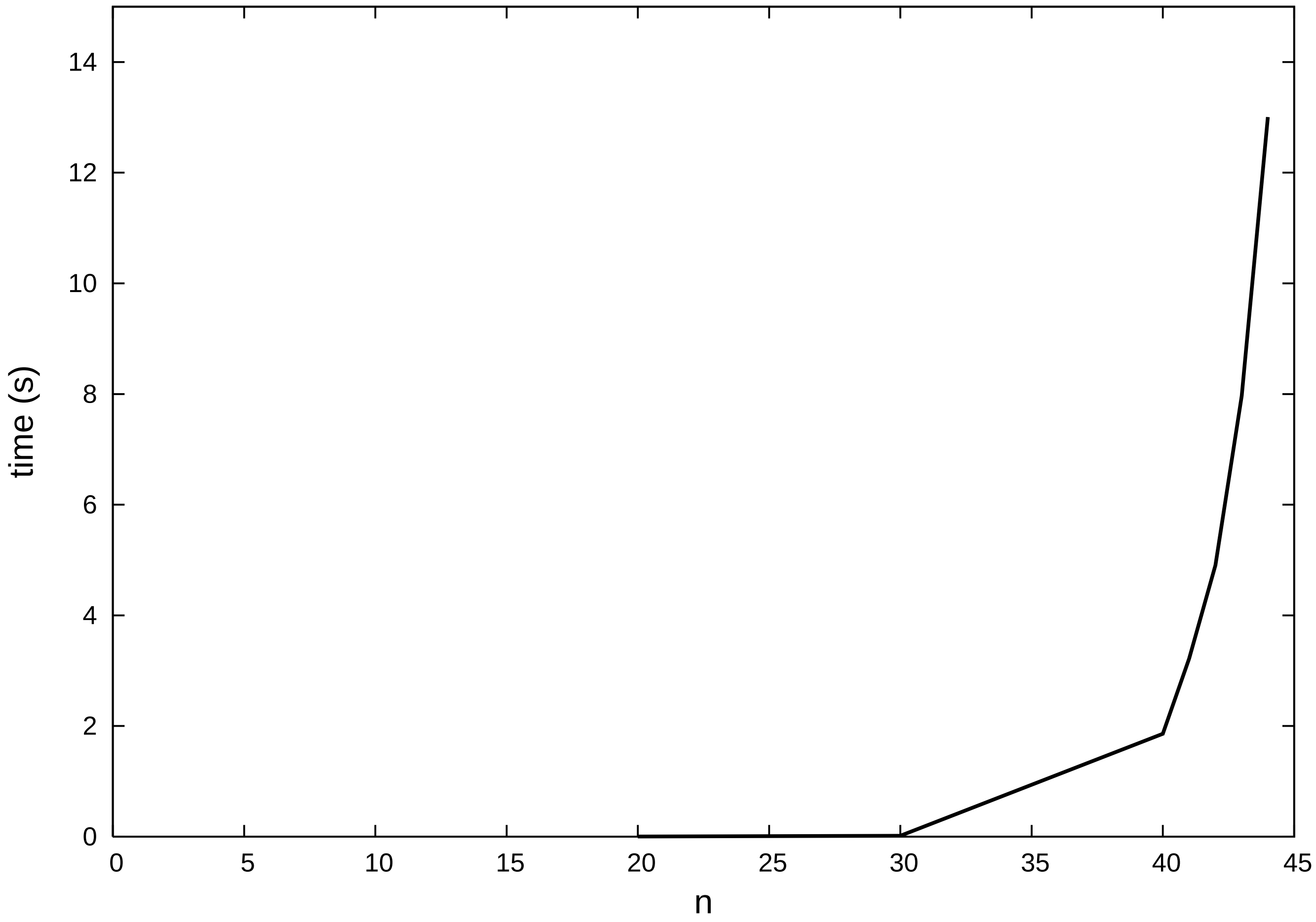
```
int Fibonacci(int n) {
    if (n <= 1) return n;
    return Fibonacci(n-1) + Fibonacci(n-2);
}
```

- A straightforward coding of the mathematical recurrence formula
- Looks benign

Run time on a 3.4GHz Intel Xeon:

n	20	30	40	41	42	43	44
Run time (s)	0.002	0.017	1.860	3.217	4.906	7.970	13.004

In graphical form...



What's going on?

We can model the run time with a recurrence relation:

- $f(n) = f(n-1) + f(n-2) + k \qquad (n > 1)$
- By induction we can prove that $f(n) \geq F_{n+1}k$
- And F_n grows exponentially: $\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \phi$ (golden ratio, ≈ 1.6)

Clearly this algorithm is impractical even for fairly modest input values, and faster computers will not help us.

- Let us say that computers double in speed each year (optimistic!)
- This has an insignificant effect on tractable range of input values.

Clearly the applicability of an algorithm is related to the growth of its cost function(s)...

Growth of functions

Most algorithms have an obvious primary input N, the size of which has direct impact on the algorithms's running time and space requirements:

- The number of items in a list that needs sorting
- The number of characters in a text string
- The number of vertices in a graph

Algorithm analysis is all about expressing the resource requirements of an algorithm in terms of N, using the simplest possible expressions that are accurate for large N.

Big-Oh notation (1)

We'd like to capture the growth of an algorithm's cost function for large N in the simplest possible expression. We use an *asymptotic notation* to capture this 'big picture' while suppressing (relatively) insignificant details.

The most common is the "big-Oh notation":

- We say that $f(N) = O(g(N))$ if there exist constants c and N_0 such that $f(N) < cg(N)$ for all $N > N_0$.
- This notation is used for a range of purposes. We will use it to provide a succinct *upper bound* on an algorithm's resource usage.
- A more succinct definition: $f(N) = O(g(N)) \Leftrightarrow \lim_{N\to\infty} \frac{g(N)}{f(N)} > 0$
- That is: f does not *dominate* (grow faster than) g.

Big-Oh notation (2)

Big-Oh notation allows us to simplify and classify cost functions according to the dominating term.

- We can ignore constant factors: $2N^3 = O(N^3)$
- We can ignore slower-growing terms: $N^3 + N^2 = O(N^3)$ (assuming $N > 1$, $N^2 < N^3$ so $N^3 + N^2 < 2N^3 = O(N^3)$)
- We can ignore bases of logarithms: $\log_x N = O(log_y N)$ for all $x, y > 1$
- In general we can classify an algorithm according to the smallest $O(g(N))$ that we can prove it is a member of

For example, consider again the recursive Fibonacci algorithm:

- $f(N) = O(\phi^N)$, where $\phi = (1 + \sqrt{2})/5$

Common growth rates

Most algorithms we will analyse have resource requirements proportional to one of the following functions:

$O(1)$	Constant (unrelated to input size)
$O(\log N)$	Logarithmic
$O(N)$	Linear
$O(N \log N)$	"Log-linear"
$O(N^k)$	Polynomial
$O(k^N)$	Exponential

Obviously these have wildly different growth rates:

$\lg N$	$\sqrt{N}$	N	$N \lg N$	N^2	N^3	2^N
3	3	10	33	100	1000	1024
7	10	100	664	10000	1000000	10^{30}

Other asymptotic notations

Big-Oh belongs to a family of asymptotic notations. Others include:

- Big-Omega provides an asymptotic lower bound:
  $f(N) = \Omega(g(N)) \quad \Leftrightarrow \quad \lim_{N\to\infty} \frac{g(N)}{f(N)} < \infty$
- Big-Theta provides an asymptotically tight bound on a function (bounded from above *and* below):
  $f(N) = \Theta(g(N)) \quad \Leftrightarrow \quad 0 < \lim_{N\to\infty} \frac{g(N)}{f(N)} < \infty$
- Clearly $f(n) = \Theta(g(N)) \ \Leftrightarrow \ f(N) = O(g(N)) \wedge f(N) = \Omega(g(N))$
  - If f is bounded from above *and* below by g, then g is an asymptotically tight bound.
- e.g., Recursive Fibonacci is $\Theta(\phi^N)$ but not $\Theta(2^N)$ (think about $\lim_{N\to\infty} 2^N/\phi^N$)

Worst, average, best costs

Many algorithms have widely varying resource usage depending on input (even for different inputs with the same 'size' N).

Worst-case analysis is based on the most unfortunate data input that might be offered to the algorithm.

Best-case analysis is based on the best-possible data input.

Average-case analysis really requires the distribution of probabilities of all different inputs. Commonly, however, all inputs are considered equally likely.

➤ Unless stated otherwise, a cost function must be true for *all* inputs.

Amortised analysis

- Amortised analysis is used for algorithms in which the cost of an operation may depend on the *history* of operations that preceded it.
- In some cases an occasional expensive operation is *amortised* across a number of subsequent operations that can be proven to run more quickly.
- Amortised complexity is the cost per operation averaged across a 'long enough' sequence of operations.
- An example we'll see later are *splay trees*.

Models of memory

- Our assumption in this course will be that computers will always have enough memory to run the algorithms we dicuss.
- We will talk about the *space complexity* of algorithms: can use this to determine if an algorithm might need a *ridiculous* amount of space.
- Stack usage is an important consideration for recursive programs.
- Paging in the OS will often turn this into an issue of speed.

Another issue is the time taken to access memory:

- Modern systems have caches which can affect the speed of a memory access by several *orders of magnitude*
- Locality of reference is increasingly important
- Our analyses will ignore these issues. A memory access is $O(1)$.

Models of arithmetic

- Assume that each arithmetic operation takes unit time ($O(1)$)
- As with memory-access time, asymptotic notation swallows any constant factors
- When analysing algorithms involving integers it is usually assumed that the algorithm will be used on fixed-width integers.
- Where this is not the case, complexity analysis may need to account for the extra work for multi-word arithmetic
- In such cases the cost function will depend not only on the input size (number of values), but also on the number of bits (or digits) needed to specify each value.

Algorithm design techniques

There are a number of well-known techniques that appear in various algorithms. We'll look at just a few of them in this course:

- Divide & conquer

- Greedy algorithms

- Dynamic programming

Divide & conquer

The classic recursive approach to solving a problem:

1. **Divide** the problem into one or more smaller pieces.
2. **Conquer** the smaller problems by recursively invoking the algorithm.
3. **Combine** the results of the smaller problems to produce a final result.

Remember that you need a base case, to avoid infinite recursion!

- The recursive Fibonacci algorithm is a (bad!) example of D&C.
  - The sub-problems are hardly smaller than the original!
- A better example is binary search on a sorted array
- Or quicksort, or mergesort, ...

Recurrence formulae (1)

As we saw with the Fibonacci example, the cost function of a divide-&-conquer algorithm is naturally expressed as a recurrence formula.

- To reason about the complexity of an algorithm we would like its cost function in a *closed* form.
  - $f(N) = \Theta(\phi^N)$ rather than $f(N) = f(N-1) + f(N-2) + k$
- Finding the closed form of a recurrence formula is not always straightforward, but there are a few standard forms that crop up repeatedly in algorithm analysis.
- $f(N) = 2f(N/2) + N$ has solution $f(N) \approx N \log_2 N$
  - An algorithm which recursively divides the problem in two halves, but must scan every input item to do so.

Recurrence formulae (2)

- $f(N) = f(N/2) + 1$ has solution $f(N) \approx \log_2 N$
  - An algorithm which recursively halves the problem size, taking unit time to divide the problem.
- $f(N) = f(\alpha N) + N$ has solution $f(N) = \Theta(N)$ if $\alpha < 1$
  - An algorithm which reduces the problem size by a fixed factor in linear time has linear complexity (although the constant of proportionality may be large!)

Greedy algorithms (1)

Many algorithms involve picking some form of *optimal* solution. Greedy algorithms proceed iteratively, always making the choice that seems best right now.

Example: fractional knapsack problem

A thief robbing a store finds n items: the ith item has value v_i and weighs w_i kilograms. The thief's knapsack can carry items up to maximum weight of W kilograms. The thief may take fractions of items.

Solution:

1. Compute value per kilogram of each item v_i/w_i.
2. Fill the knapsack with as much as possible of each item in descending order of value-by-weight.

Greedy algorithms (2)

The fractional knapsack problem exhibits *optimal substructure*

- The solution for any particular size of knapsack is contained within the solution for all larger knapsacks.
- Hence it always makes sense to steal as much as possible of the most valuable remaining item.

The main source of optimal greedy algorithms is in the study of graph problems:

- Minimum spanning tree, shortest path, ...
- More on these next year!

Dynamic programming

- Similar to divide-&-conquer, dynamic programming solves problems by combining results of smaller subproblems.
- It is used where the subproblems overlap: dynamic programming remembers the partial solutions so each subproblem is solved only once.
- Two varieties:
  - Bottom-up: solve increasing large subproblems.
  - Top-down (*memoisation*): recursively decompose the problem, but reuse solutions to previously-solved subproblems.
- It only works for 'integer' problems, where the number of unique subproblems is bounded by the size of the input
  - If the problem is parameterised by real numbers, there may be an infinite number of subproblems.

Dynamic programming: examples

We can use dynamic programming to come up with a much faster algorithm for the Fibonacci sequence:

```
int fib[] = new int[N]; fib[0] = 0; fib[1] = 1;
for ( i = 2; i < N; i++ ) fib[i] = fib[i-1] + fib[i-2];
return fib[N];
```

Another example is the *0-1 knapsack problem*. It is identical to the fractional version except that the thief can take only whole items.

- A greedy algorithm does not work!
- Consider 2 items: 10kg worth £15, and 20kg worth £20
- Solution for 10kg knapsack is not part of solution for 20kg knapsack

0-1 knapsack problem

- Let C be the knapsack capacity and I be the set of N items, each of which is a pair (w, v) where w is the item's weight and v its value.
- We'll consider the items in some predetermined total order, such that $I_0 = I$; $I_k = \{w_k, v_k\} \cup I_{k+1}$; $I_N = \{\}$.
- Then we have maximum knapsack value $M(C, I)$ defined as:

$$
\begin{array}{lcll}
M(c, I_k) & = & \text{max}(\, M(c, I_{k+1}),\ M(c - w_k, I_{k+1}) + v_k\,) & (w_k \leq c) \\
M(c, I_k) & = & M(c, I_{k+1}) & (w_k > c) \\
M(c, \{\}) & = & 0 &
\end{array}
$$

- The recursive implementation has exponential complexity
- Top-down dynamic programming reduces complexity to $O(CN)$

Data structures

Many algorithms organise data in a very specific way that aids the computation.

- These arrangements are called *data structures*
- We'll see a range of clever data structures when we look at searching
- But first we need to look at some of the basics of data organisation

Scalar types All general-purpose programming languages have a range of simple (scalar) data types, including booleans, integers and floating-point values.

Array types Used when multiple values need to be recorded, each of the same type. Component values are accessed by their position in the array $(0..n-1)$ and can typically be accessed in $O(1)$ time.

Record data types

Often we want to aggregate a number of related data values. Most languages allow us to define compound (record) data types.

For example, in Java:

```
class Employee {
    String name;
    Address address;
}
```

Or in C:

```
struct Employee {
    char *name;
    struct Address address;
}
```

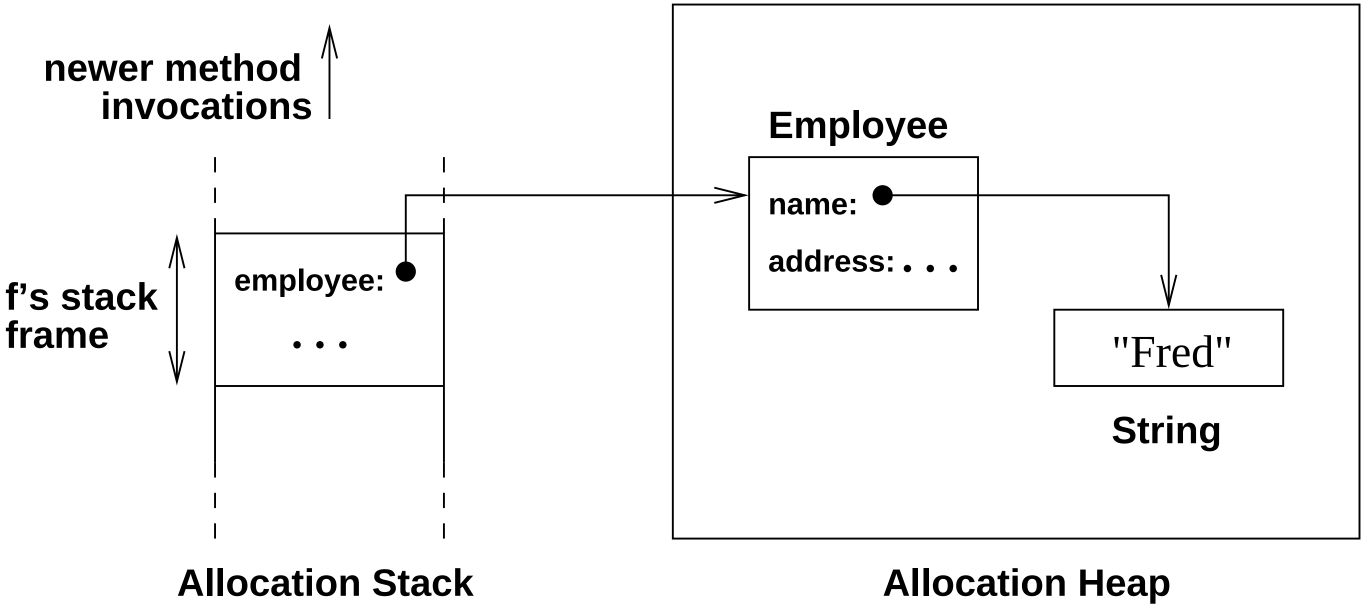
Allocating and referencing data

Data values need to be placed somewhere in the computer's memory. This allocation problem is addressed in a number of ways.

- Data items declared within a function/method are allocated on a *stack*. This is highly efficient, but the variable is only usable until the end of the function.
  - The 'last-in-first-out' behaviour of a stack neatly models the nested structure of method calls in a computer program.
- Data items declared outside any method or class are *global*. They are allocated when the program is loaded into memory.
- Data items may also be *dynamically allocated*. In Java, all objects are dynamically allocated and automatically freed.

Dynamic allocation (1)

```
void f(void) {
    Employee employee = new Employee();
    employee.name = "Fred";
}
```



Dynamic allocation (2)

Note that the stack is simply a contiguous region of memory, growing upwards in the illustration.

- Grows upward on each method invocation
- Shrinks downward on each method return

Objects are not allocated on the stack.

- Object variables are actually *references* (or *pointers*) into the heap.
- Objects are always allocated on the heap, using `new()`.
- Java automatically frees (*garbage collects*) unused objects.
- Other languages have similar constructs (C's `malloc`/`free`).

Pointers play a critical role in dynamic data structures.

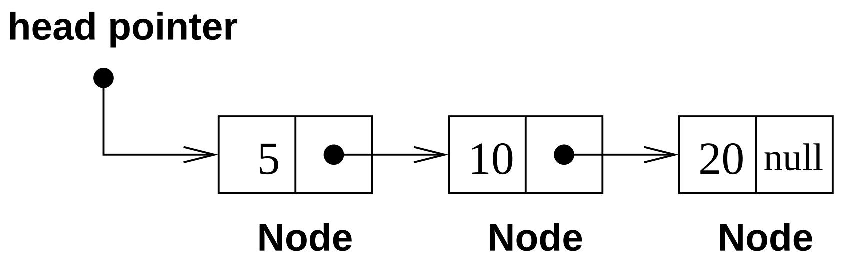
Linked lists

How can we store a dynamic collection of data records in a computer program?

- Could use an array, but it's fixed size.
- May be better to dynamically allocate each record and somehow link them together into a set.

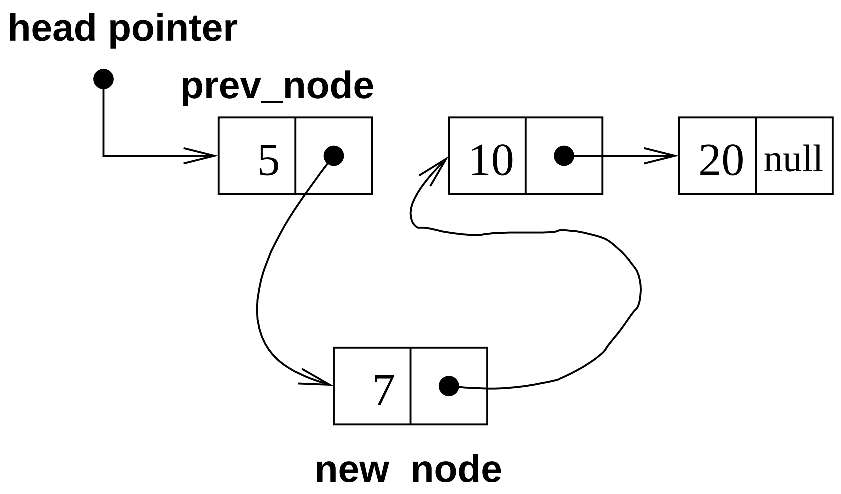
This concept of a linked dynamic data structure is extremely common. The simplest such structure is the *linked list*.

```
class Node {
    int value;
    Node next;
}
```



Linked lists: insertion

```
void insert(Node new_node, Node prev_node) {
    new_node.next = prev_node.next;
    prev_node.next = new_node;
}
```



How to insert at head of a linked list?

➤ Often have a sentinel head node to avoid special case

Linked lists: variations

Clearly we can easily traverse the entire set of nodes in a list, and easily insert new nodes (if we know the predecessor). But what if:

- We want to be able to scan the list in both directions?
- Delete an item without knowing its predecessor?

Variations therefore include:

- Doubly-linked list (point to successor and predecessor)
- A list with both a head and tail pointers
- A circular linked list

The best choice is dependent on application. Often a specifically tailored implementation is 'inlined' into application code.

Abstract data types

Without some care it is very easy to get tied up in the details of the data structures and algorithms that you are implementing, and lose sight of the requirements of your application.

- Makes it hard to change data structures and algorithms.
- Makes a program hard to understand and maintain.

Abstract data types provide a way to reason about *what* it is you are trying to do, rather than the details of *how*. They can be applied at various levels.

- You might divide your application into coarse-grained ADTs or modules, corresponding to concepts very close to the problem domain.
- These may decompose into more primitive ADTs which still abstract away from the very lowest-level implementation details.
- Software engineering is largely about the art of doing this well.

The *stack* abstract data type

A simple example:

`s = empty_stack()`	returns a new, empty stack
`push(s, x)`	adds `x` to the top of the stack
`x = pop(s)`	removes and returns the topmost item

The object-oriented features of Java make it easy to represent ADTs:

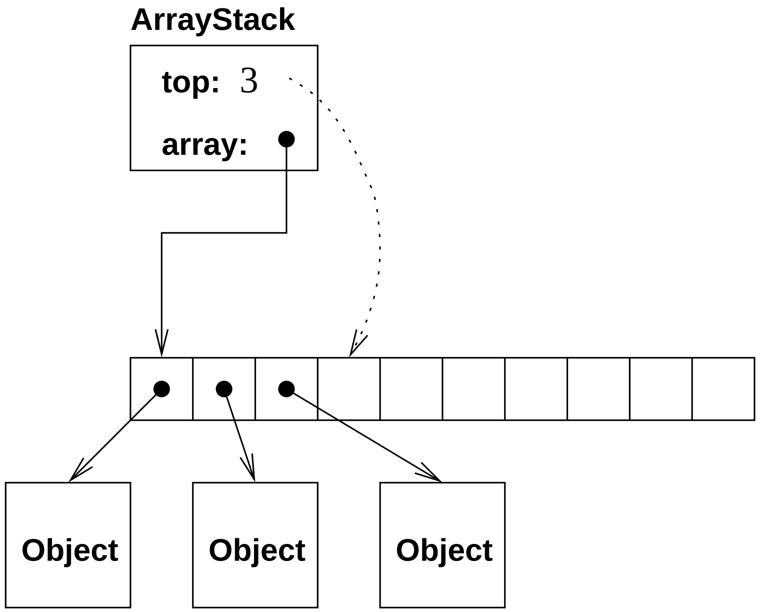
```
interface Stack {
    void push(Object x);
    Object pop(void);
}
```

- An implementing class's constructor is equivalent to `empty_stack()`.
- The interface's methods take the stack object as an implicit parameter.
- Works on any data value that can be encapsulated in an `Object`.

Stack based on array

- The stack ADT may be implemented by a variety of data structures.
- An array can be used if it is known how many items may be stored in the stack at a time.

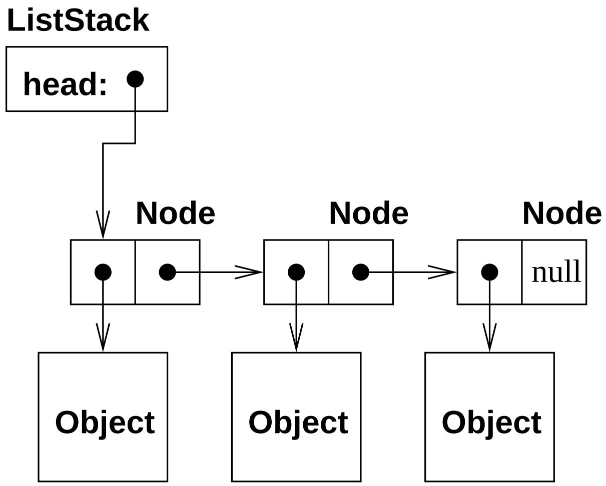
```
class ArrayStack implements Stack {
    Object[] array;
    int top;
    void ArrayStack(int size) {
        array = new Object[size];
    }
    void push(Object x) {
        array[top++] = x;
    }
    Object pop(void) {
        return array[--top];
    }
}
```



Stack based on linked list

- A linked list provides a dynamically-sized alternative.

```
class Node {
    Object item;
    Node next;
}
class ListStack implements Stack {
    Node head;
    void push(Object x) {
        Node new_node = new Node();
        new_node.item = x;
        new_node.next = head;
        head = new_node;
    }
    Object pop(void) {
        Node top_node = head;
        head = top_node.next;
        return top_node.item;
    }
}
```



The *queue* abstract data type

`q = empty_queue()`	returns a new, empty queue
`add_tail(q, x)`	adds `x` to the tail of the queue
`x = remove_head(q)`	removes and returns the head of the queue

- The semantics are quite different to a stack: items are removed in FIFO order rather than LIFO.
- Again, various different implementations are possible
  - It is worth trying to implement array- and list-based variants as an exercise (think of the array as a circular buffer)
- Other queuing related ADTs are possible (e.g., deque)

The *table* abstract data type

`t = empty_table()`	returns a new, empty table
`set(t, k, x)`	adds the mapping `k`$\rightarrow$`x`
`x = get(t, k)`	returns the item `x` that `k` maps to

- Rather like the ADT of the search structures we will be looking at later
- Sorted linked list: `get` and `set` are $O(N)$ (obvious?)
- Sorted array: `get` is $O(\log N)$, but `set`?

```
Object get(int k) {
    int m, l = 0, r = nr_items-1;
    while ( true ) {
        if ( l >= r ) return null;
        m = (l + r) / 2;
        if ( a[m].key == k ) return a[m].item;
        if ( a[m].key < k ) l = m+1; else r = m-1;
    }
}
```

Sorting

- Sorting is a big set-piece topic in any fundamental algorithms course.
- Given an input array of N numbers, sort them into ascending order
- The problem statement is easy, but there are a wide range of solutions
- Arranging data into order is an important problem, as you might expect
  - Ease of searching (see the array-based table implementation!)
  - Fast in-order scanning (good for external storage)
  - Many other application-specific uses
- The algorithms we'll look at divide into three broad categories
  - The simple comparison-based sorts (quadratic complexity)
  - The fast comparison-based sorts (usually $O(N \log N)$)
  - Sorts which are not based on comparing key values

Sorting data records

- We'll be looking at sorting arrays of integers, but that's not very interesting as an end in itself
- In practise the array will usually contain compound data records
- One field in each data record will contain the *key value*
- It's that field which which we use to rearrange the records
  - Usually, but not always, it's an integer field
  - If not, need to define an ordering to use comparison-based sorts
- Most sorts are based on exchanging the positions of items in the list
  - Define `exchange(a,i,j)` to swap items at positions `i,j` in array `a`
- Exchanging big data records is cheap if the array holds only *references*
  - Remember that will always be the case in Java

Stable sorts

A sorting algorithm is said to be *stable* if it preserves the relative order of items with identical keys.

- Consider sorting an alphabetised list of student records by exam grade.
  - We would like the sort to maintain alphabetical order for students with the same grade.

Adams	II.1		Jones	I		Jones	I
Barker	II.1	$\longrightarrow$	Adams	II.1	**NOT**	Smith	II.1
Jones	I		Barker	II.1		Adams	II.1
Smith	II.1		Smith	II.1		Barker	II.1

- Important for multistage sorting (e.g., sort by name *then* by grade)
- We can work around an unstable sorting algorithm by extending the keys to include the position in the original array.

Insertion sort (1)

This algorithm iterates over each list item. The ith iteration places the ith item into its correct position among the first i items.

i=1:	9 7 4 8 2 3	i=4:	4 7 8 9 2 3
i=2:	7 9 4 8 2 3	i=5:	2 4 7 8 9 3
i=3:	4 7 9 8 2 3	final:	2 3 4 7 8 9

```
for ( i = 0; i < N; i++ )
    for ( j = i-1; (j >= 0) && (a[j] > a[j+1]); j-- )
        exchange(a, j, j+1);
```

- Insertion sort is **stable**!

Insertion sort (2)

- We often evaluate the efficiency of a sorting algorithm in terms of the number of *comparisons* (C) and *exchanges* (X) that it performs.
- Worst case for insertion sort is that the inner loop runs until $j = 0$. This will happen if the array is originally in reverse order.

$$\begin{aligned} C \;=\; X &= \sum_{k=0}^{N-1} k \\ &= \tfrac{1}{2}(N-1)(N-2) \\ &= \Theta(N^2) \end{aligned}$$

- Hence insertion sort has *quadratic complexity* in the worst case.
- But it has an interesting best case: what if the array is already sorted?
  - One comparison and no exchanges per iteration.
  - Useful for 'finishing' a sort started by a more efficient algorithm.

Selection sort (1)

- Like insertion sort, it iterates N times.
- On the ith iteration, the next smallest item in the list is *exchanged* to its final position on the sorted result.
- On each iteration, the remaining $N - i$ items in the list are scanned to find the smallest.

i=0:	9 7 4 8 2 3	i=3:	2 3 4 8 9 7
i=1:	2 7 4 8 9 3	i=4:	2 3 4 7 9 8
i=2:	2 3 4 8 9 7	i=5:	2 3 4 7 8 9

- The exchanges make selection sort **unstable**: consider list $(2_a, 2_b, 1)$

Selection sort (2)

Pseudocode:

```
for ( i = 0; i < N; i++ ) {
    for ( j = min = i; j < N; j++ )
        if ( a[j] < a[min] ) min = j;
    exchange(a, i, min);
}
```

- The number of comparisons in selection sort is always $\frac{1}{2}N(N-1)$
- Hence the best- and worst-case complexity is $\Theta(N^2)$.
- However, it never performs more than N exchanges.
  - Useful when exchanges are expensive?
- Later we'll see a variant with a much faster find-minimum sub-algorithm.

Bubble sort (1)

- The inner loop iterates over each adjacent pair of items
- Exchanges them if they are out of order (bubble sort is **stable**)

i=0:	9 7 4 8 2 3	i=3:	7 4 8 9 2 3
i=1:	7 9 4 8 2 3	i=4:	7 4 8 2 9 3
i=2:	7 4 9 8 2 3		7 4 8 2 3 9

- Need to scan $N-1$ times to guarantee the list is fully sorted
  - Consider a list in reverse order
- Hence a double loop, each iterates $N-1$ times: $\Theta(N^2)$.

Bubble sort (2)

Pseudocode:

```
for ( i = 0; i < N-1; i++ )
    for ( j = 0; j < N-1; j++ )
        if ( a[j] > a[j+1] ) exhange(a, j, j+1);
```

- Often rather fewer than $N-1$ iterations are required.
- A common optimisation is to finish early if no items are exchanged during any complete pass over the list.
  - But slows down the worst case due to extra bookkeeping overheads.
- However, even when this optimisation is of benefit, insertion sort will generally do even better.
  - If you remember one simple sorting algorithm, make it insertion sort rather than bubble sort!

Shell sort (1)

- Insertion sort has good performance when the list is not too random
- How can we improve its worst-case performance?
  - Try to *roughly* sort the list
  - Perform preliminary insertion sorts on subsets of the list
  - 'Stride-s' sort acts on s subsets of the array:
  - Subset k $(0 \leq k < s)$ contains items $k, s+k, 2s+k, \ldots$
- s starts large (e.g., size of array) and shrinks down to 1 (which is a normal insertion sort).
  - A good sequence is $\ldots, 40, 13, 4, 1$ where $s_{i-1} = 3s_i + 1$
- Analysis is hard. Run time is $O(n^{1.5})$ but a tight bound is not known. In practise Shell sort usually beats $O(n \log n)$ algorithms!

Shell sort (2)

Pseudocode:

```
for ( s = initstride(N); s > 0; s /= 3 )
    for ( i = 0; i < N; i++ )
        for ( j = i-s; (j >= 0) && (a[j] > a[j+s]); j -= s )
            exchange(a, j, j+s);
```

- Notice how the stride-s sorts are interleaved.
  - Rather than sorting each subset in turn, we make one pass through the array (on index i)
- Notice the similarity to normal insertion sort.
- Strange that this *triple loop*, which superficially appears to do more work, performs so much better than normal insertion sort!
- One downside: shell sort is **unstable**

Quicksort

A divide-&-conquer sorting algorithm, also known as 'partition-exchange'.

Divide Separate list items into two partitions. The left partition contains only items $\leq$ those in the right partition.

Conquer Recursively invoke quicksort on the two partitions.

```
void qsort(int[] a, int l, int r) {
    if ( l >= r ) return;
    m = partition_exchange(a, l, r);
    qsort(a, l, m-1);
    qsort(a, m+1, r);
}
```

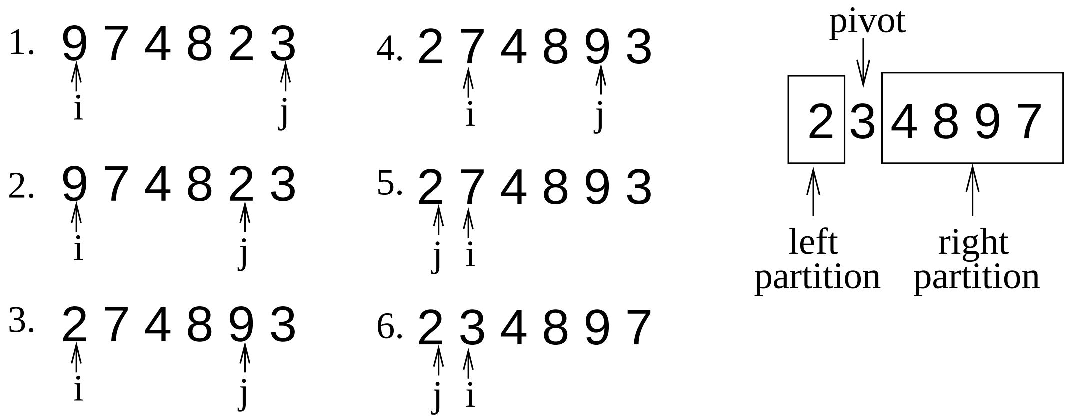
- Partition-exchange picks a pivot value to partition around.
- The pivot value is always placed in its final sorted position m.

Quicksort: partitioning

- Any value in the array can be chosen as pivot.
  - Items in left partition are $\leq$ pivot
  - Items in right partition are $\geq$ pivot
  - We *hope* it is close to the median value
  - Leads to equal sub-partitions and maximal efficiency
- The hard work is involved in partitioning the array, avoiding problems with boundary cases (e.g., all key values the same).
- There are a variety of efficient methods
- The one we'll look at uses two indexes i and j which start at opposite ends of the initial sub-array (l and r respectively)
- The rightmost item (index r) is used as pivot...

Quicksort: partitioning example

- Set indexes i and j to l and r respectively
- Increment i until $a[i] > a[r]$ ($a[r]$ is the pivot value).
- Decrement j until $a[j] < a[r]$.
- Exchange $a[i]$ and $a[j]$. Continue until indexes cross.
- Finally, exchange $a[r]$ with $a[i]$.



Quicksort: average- and worst-case costs

Assuming equal-sized subpartitions, quicksort has cost
$f(N) = 2f(N/2) + \Theta(N) = \Theta(N \log N)$

But this depends on good choice of pivot! The worst case is picking the smallest or largest value as pivot: $f(N) = f(N-1) + \Theta(N) = \Theta(N^2)$

Our current strategy, picking the last item as pivot, is *pessimal* for arrays that are sorted or in reverse order!

A common strategy for avoiding the worst case is to pick a median-of-three as pivot: the first, middle and last items in the array.

- Avoids bad performance for nearly-sorted lists
- But doesn't *guarantee* $\Theta(N \log N)$ performance.

Quicksort: space complexity

- Because Quicksort is a recursive algorithm, its space complexity is not $O(1)$.
- Each recursive invocation requires a fixed amount of stack space, so Quicksort's space complexity is proportional to maximum depth of recursion.
- In the worst case that is $\Theta(N)$ However, a simple change to Quicksort ensures that the worst case is $\Theta(\log N)$:

```
while ( l < r ) {
    m = partition_exchange(a, l, r);
    if ( (m-l) < (r-m) ) { /* recurse on the *smaller* partition */
        qsort(a, l, m-1); l = m+1;
    } else {
        qsort(a, m+1, r); r = m-1;
}}
```

Priority queues

A data structure that allows retrieval of the smallest value it contains:

`q = empty_pqueue()`	returns a new, empty priority queue
`add(q, x)`	adds `x` to the priority queue
`x = remove_min(q)`	removes and returns the smallest item

For example:

```
add(q,4)
add(q,2)
add(q,7)
remove_min(q) = 2
remove_min(q) = 4
remove_min(q) = 7
```

Priority queues: applications

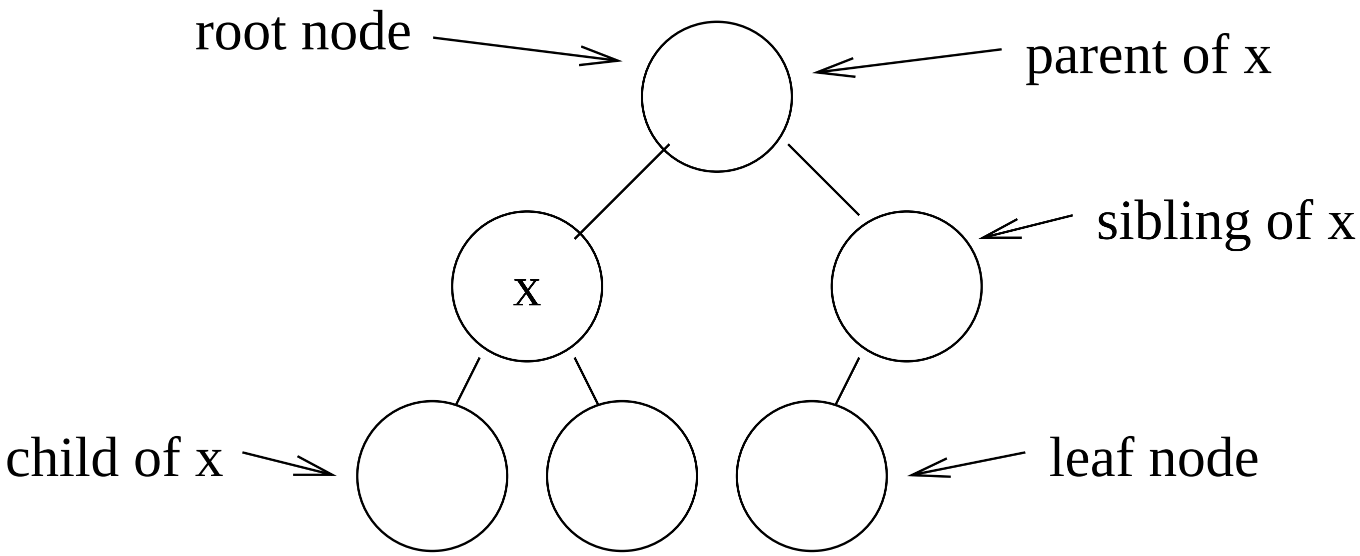
- Useful for retrieval based on time (what is the *next* event in my calendar?)
- Useful for sorting!

```
q = empty_pqueue();
for ( i = 0; i < N; i++ )
    add(q, a[i]);
for ( i = 0; i < N; i++ )
    a[i] = remove_min(q);
```

- Clearly the efficiency depends on the costs of the priority queue implementation.

Tree structures

Trees are the most common non-linear data representation in computing.

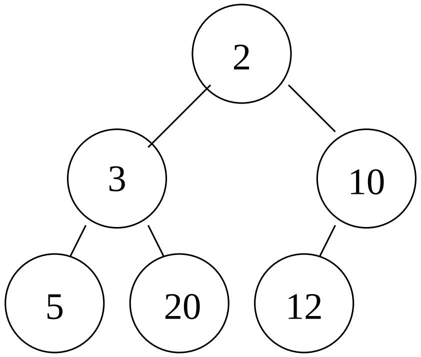


- This is a *binary tree*: each node has at most two children.
- It is a *complete* binary tree: every level is filled except the bottom level (which is filled left-to-right)

Heaps

Tree structures are interesting when there is some ordering relationship among the nodes.

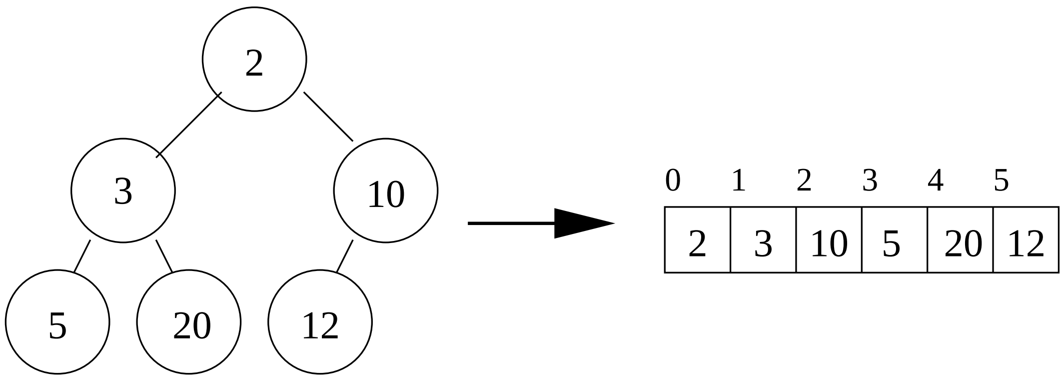
- The classic example is a binary search tree.
- A binary tree is a *heap* if it is *complete* and every node obeys the *heap property*.
  - The value of each node is less than or equal that of its children.
  - Thus the root node contains the smallest value in the heap.



Array representation of a binary tree

- We can represent a *complete* binary tree in an array.
- The 'node' at array index i has children at positions $2i + 1$ and $2i + 2$
- The parent is at location $\lfloor \frac{1}{2}(i - 1) \rfloor$

An example array representation of a heap:

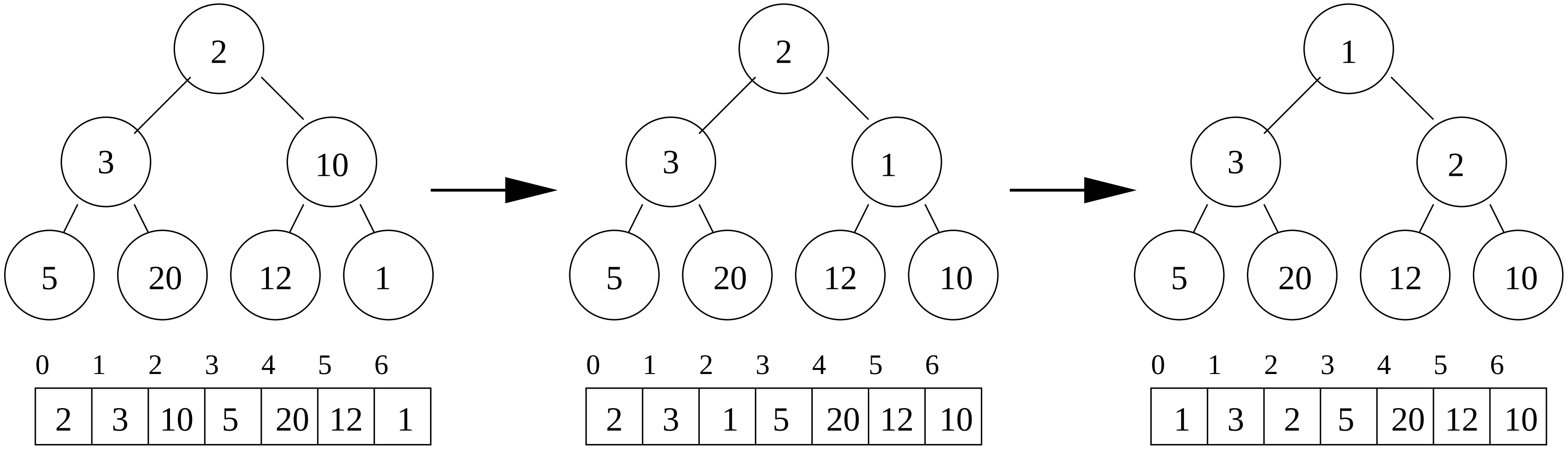


- The array is filled by scanning the tree left-to-right and top-to-bottom

Heap: adding an item

To add an item to an array-based heap, the obvious place to place it is at the 'bottom right' of the tree (the end of the array).

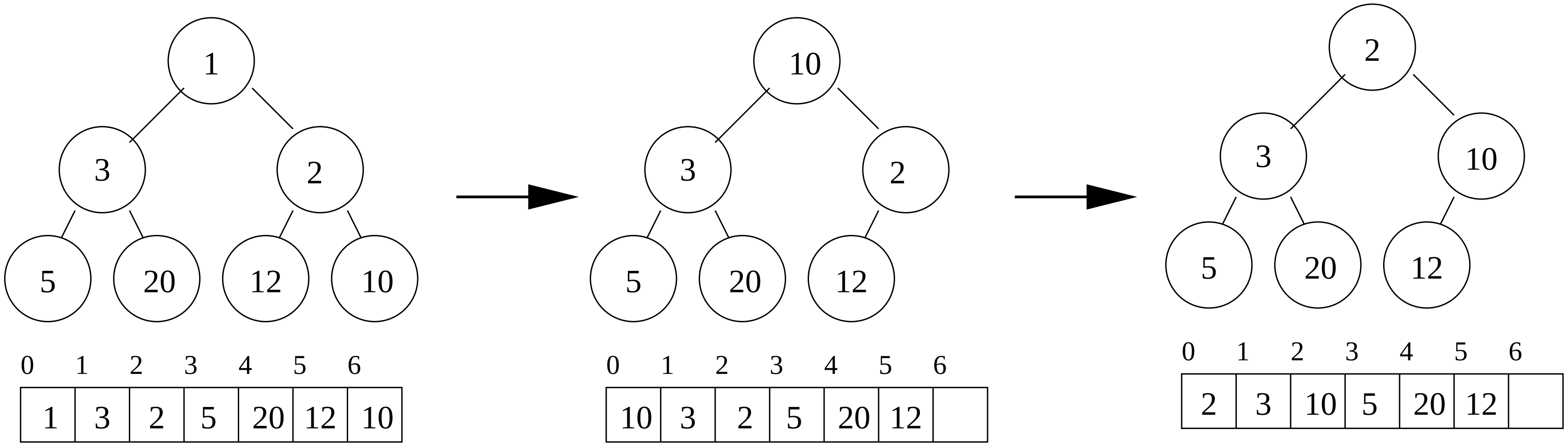
- But now the heap property may no longer hold.
- We can fix this by 'floating' the new item up to its correct level
  - Repeatedly compare with parent: exchange if out of order



Heap: removing minimum item

If we remove the minimum item (at index 0) what do we place in the hole?

- The item from the bottom-right (at the end of the array)
- Then 'sink' the item to its correct level to restore the heap property
  - Always exchange with the *smaller* child value



Heap: analysis (1)

The worst-case time complexity of adding or removing an item from a heap of N items is $\Theta(\log N)$

- The height of a complete binary tree is $\lceil \log_2(N+1) \rceil$
- Each insert/delete operation may have to float/sink an item $\Theta(\log N)$ times to restore the heap property

How long to create a heap by adding N items to an empty heap?

- Time cost is proportional to $\log_2 1 + \ldots + \log_2 N \; = \; \sum_{k=1}^{N} \log_2 k$
- No term is greater than $\log_2 N$ and so $\sum_{k=1}^{N} \log_2 k \; \leq \; N \log_2 N$
- Thus the cost is $O(N \log N)$

Heap: analysis (2)

We can also prove that $\sum_{k=1}^{N} \log_2 k \;=\; \Omega(N \log N)$:

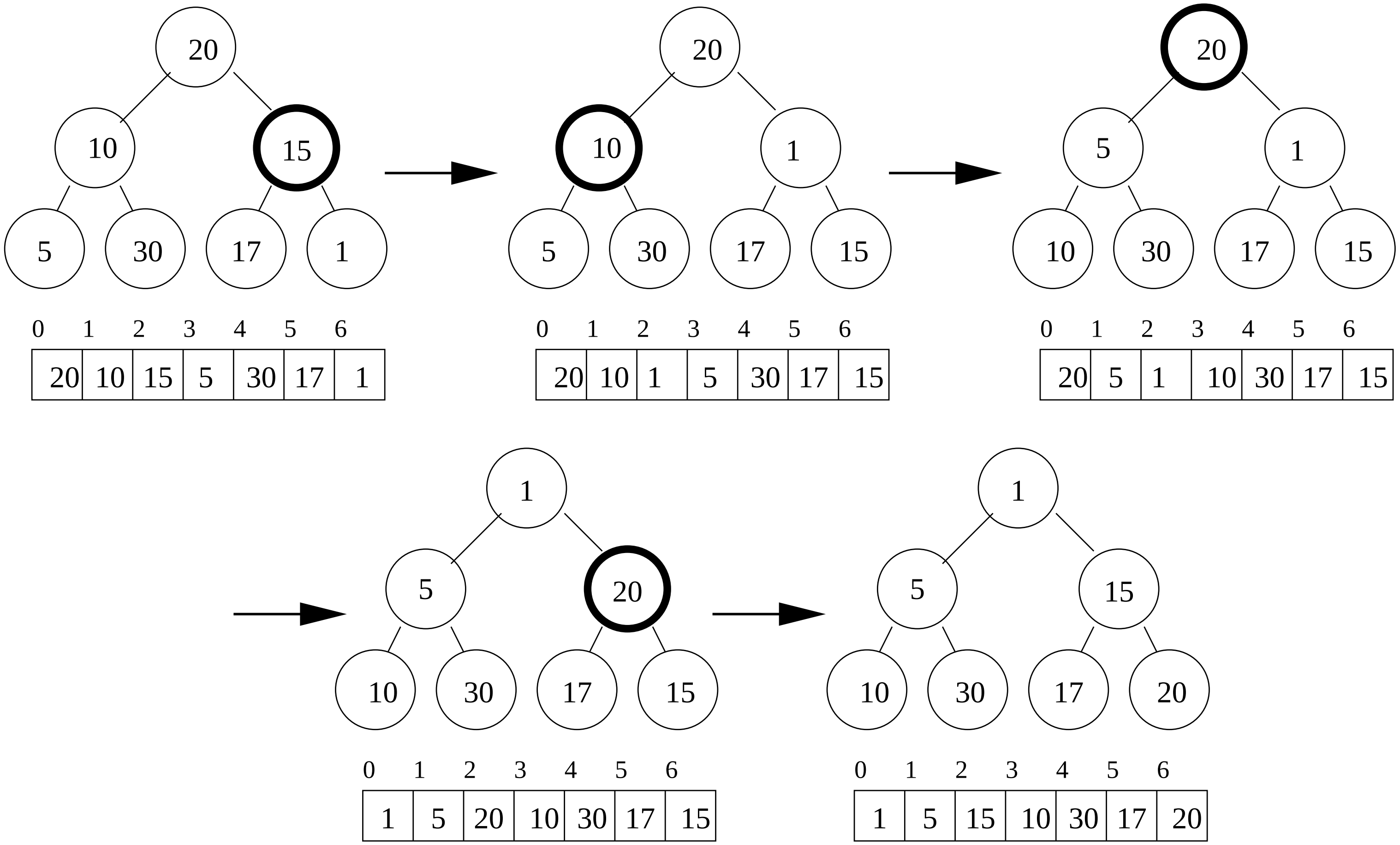
$$
\begin{aligned}
\sum_{k=\frac{N}{2}}^{N} \log_2 k &\geq \tfrac{N}{2}(\log_2 N - 1) \quad \text{(since } \log_2 \tfrac{N}{2} \;=\; \log_2 N - 1\text{)} \\
\sum_{k=1}^{N} \log_2 k &= \textstyle\sum_{k=1}^{\frac{N}{2}-1} \log_2 k + \sum_{k=\frac{N}{2}}^{N} \log_2 k \\
&\geq \textstyle\sum_{k=1}^{\frac{N}{2}-1} \log_2 k + \frac{N}{2}(\log_2 N - 1) \\
&\geq \tfrac{N}{2}(\log_2 N - 1) \quad (= \Theta(N \log N)) \\
&= \Omega(N \log N)
\end{aligned}
$$

Combined with the $O(N \log N)$ result, we can see that this is also an *asymptotically tight* bound ($\Theta(N \log N)$)

Bottom-up heapification

- We can do better than $\Theta(N \log N)$ by heapifying 'bottom up'
  - Builds a heap from an arbitrary array *inductively* and *in place*
- Start at the bottom-right (far end) of the heap (array) and work left
- Treat each node as the root of a sub-heap
  - Left and right subtrees are already valid k-level heaps
  - So sink the root node to its correct level to make a $k+1$-level heap
- Creates a valid heap in $\Theta(N)$ time
  - Most of the sink operations are on small heaps
  - The few $\log N$ sink operations do not dominate

Bottom-up heapification: example



Bottom-up heapification: linear-time proof

Consider the N-item array as a complete binary tree of height h

- An item k hops from the root can only sink $h - k - 1$ levels

Worst-case number of comparisons summed across all down-heap operations: $C = \sum_{k=0}^{h-1}(h - k - 1) \cdot 2^k$

- Each k represents all the nodes k hops from the root

Change of variable ($i = h - k - 1$):

$$\begin{array}{rcl} C & = & \sum_{i=0}^{h-1} i \cdot 2^{h-i-1} \\ & = & 2^{h-1} \sum_{i=0}^{h-1} i/2^i \\ & \leq & 2^{h-1} \cdot 2 \qquad \text{(!!!)} \\ & = & 2^h \end{array}$$

Since $h = O(\log N)$ we have $C = 2^{O(\log N)} = O(N)$

Bottom-up heapification: the tricky summation

$$S = \sum_{i=0}^{h-1} i/2^i \;\leq\; \sum_{i=0}^{\infty} i/2^i \;=\; \sum_{j=1}^{\infty} \sum_{i=j}^{\infty} 1/2^i$$

$$\begin{array}{rcccccccc}
\sum_{i=0}^{\infty} i/2^i & = & \frac{1}{2^1} & + & \frac{1}{2^2} & + & \frac{1}{2^3} & + & \dots \\
 & & & + & \frac{1}{2^2} & + & \frac{1}{2^3} & + & \dots \\
 & & & & & + & \frac{1}{2^3} & + & \dots
\end{array}$$

$$\begin{aligned}
2S - S &\leq \sum_{j=1}^{\infty} \sum_{i=j}^{\infty} 1/2^{i-1} - \sum_{j=1}^{\infty} \sum_{i=j}^{\infty} 1/2^i \\
&= \sum_{j=1}^{\infty} \left(\sum_{i=j-1}^{\infty} 1/2^i - \sum_{i=j}^{\infty} 1/2^i\right) \\
&= \sum_{j=1}^{\infty} 1/2^{j-1} \\
&= \sum_{j=0}^{\infty} 1/2^j \\
&= 2
\end{aligned}$$

Heapsort (1)

Haapsort is a selection sort that uses a heap for efficient selection. Recall the priority-queue-based pseudocode presented earlier:

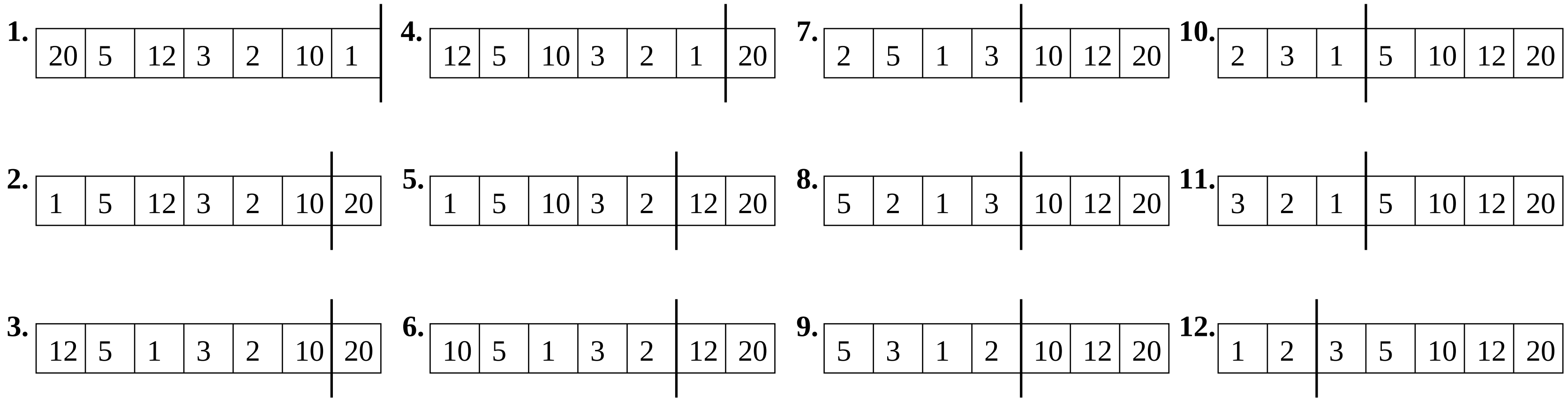
```
q = empty_heap_pqueue();
for ( i = 0; i < N; i++ )
    add(q, a[i]);
for ( i = 0; i < N; i++ )
    a[i] = remove_min(q);
```

- Each stage above has cost $\Theta(N \log N)$. Thus heapsort has a better worst-case asymptotic cost than quicksort.
- But it does require an auxiliary N-item array to hold the heap. Thus space complexity is $\Theta(N)$.
- We can do better...

Heapsort (2)

The key optimisation is to maintain the heap *in place* inside the input array.

- We can use bottom-up 'heapify' to build the heap in $\Theta(N)$ time
- Note that we build a *max-heap* not a min-heap
- We can then work backwards through the array, repeatedly extract the maximum item and place it in its correct location.



Heapsort (3)

- The cost of creating the heap is $\Theta(N)$ (bottom-up heapify)
- The cost of the selection phase is $\Theta(N \log N)$ (because most of the `remove_max` operations have $\Theta(\log N)$ cost.
- So the total cost is $\Theta(N \log N)$
- And it achieves this performance with $\Theta(1)$ workspace
- In practise heapsort does rather more work than quicksort, so it's slower for most inputs
  - Maintaining the heap property requires a significant number of compares and exchanges
- But heapsort does better in the worst case for large inputs

Mergesort: introduction

The final comparison-based sort we will look at is another device-&-conquer algorithm.

Conquer Recursively invoke mergesort on the two halves of the array.

Combine Merge the two halves together into a final result.

```
void merge_sort(int[] a, int l, int r) {
    int m = (l+r)/2;
    if ( l >= r ) return;    /* base case */
    merge_sort(a, l, m);     /* conquer left  */
    merge_sort(a, m+1, r);   /* conquer right */
    merge(a, l, m, r);       /* combine */
}
```

- How do we merge two sorted lists into a single sorted list?

Merging two sorted arrays

Merging two input arrays into a separate output array is simple. Consider input arrays $a[M]$ and $b[N]$ and output array $c[M+N]$:

```
i = j = k = 0;
while ( (i < M) && (j < N) )
    if ( a[i] <= b[j] ) c[k++] = a[i++];
    else                c[k++] = b[j++];
while ( i < M )
    c[k++] = a[i++];
while ( j < N )
    c[k++] = b[j++];
```

At each step we pick the next item from a or b that is smaller. When one of the arrays is exhausted we simply take all remaining items from the non-empty one.

Merging for mergesort

For mergesort we want a merge operation that *appears* to work in place.

- Copy the two sub-arrays into temporary workspace
- Then merge them back into the original array
- Results in $\Theta(N)$ space overheads and copying overheads

We can do a bit better by copying only the left sub-array

- The merge will never overwrite items from the right sub-array before they have been consumed by the merge
- Halves the required workspace and copying overheads

Mergesort: analysis

The running time of mergesort is independent of the distribution of input values: $f(N) = 2f(N/2) + \Theta(N) = \Theta(N \log N)$

However, the copying overhead to temporary workspace means that mergesort is generally slower than quicksort or heapsort.

- The $N/2$ units of temporary workspace must also be considered.

The sequential access behaviour of mergesort does have some advantages:

- Unlike quicksort and heapsort, mergesort is **stable**
- It's the basis for most external sorting methods
- It can easily be applied to linked lists, although care is required to avoid lots of list walking. Linked-list implementations usually have $\Theta(1)$ space complexity.

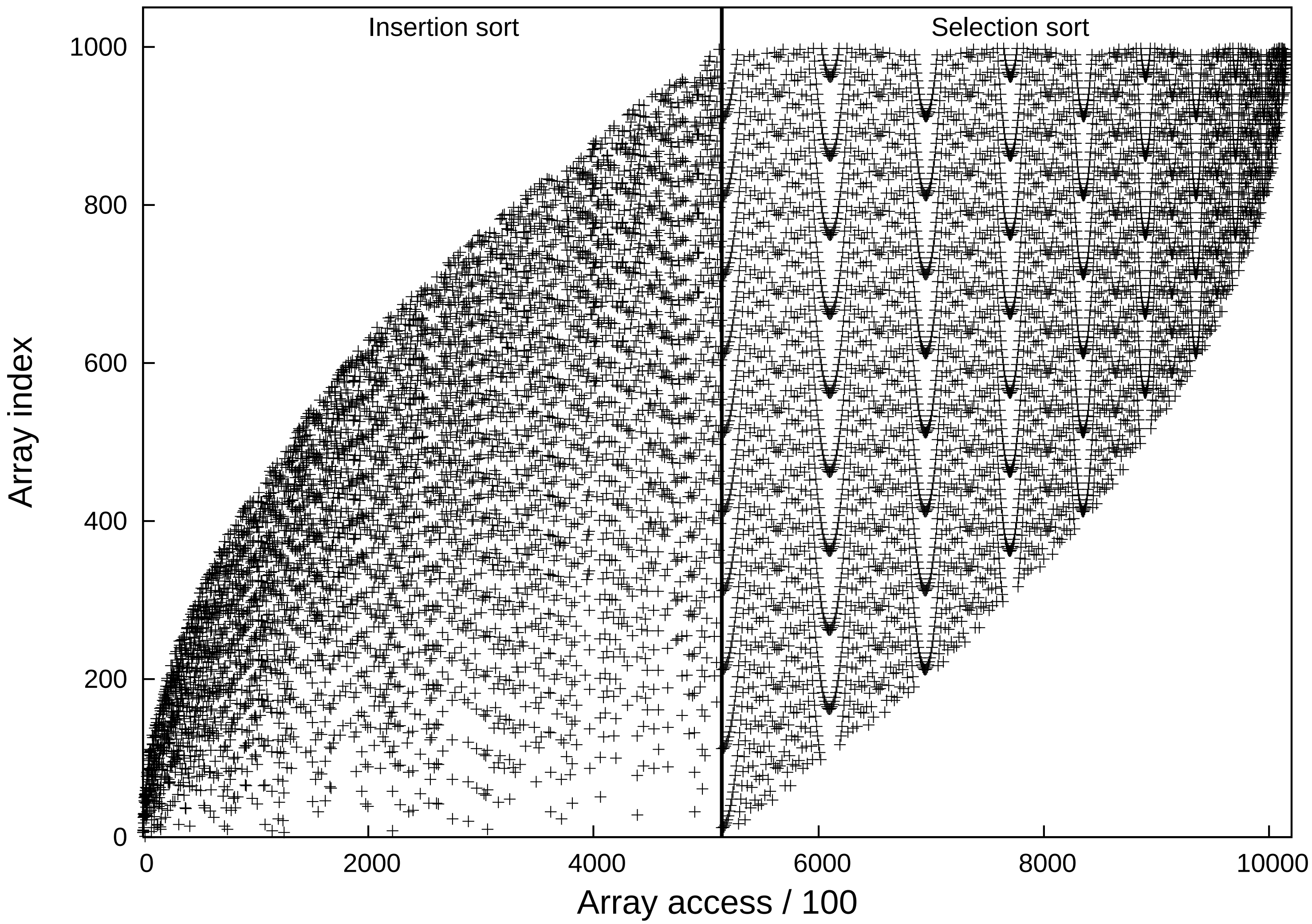
Comparison-based sorts: running times

Running times when sorting a vector of 100,000 integers using seven different methods and four different settings of the initial data.

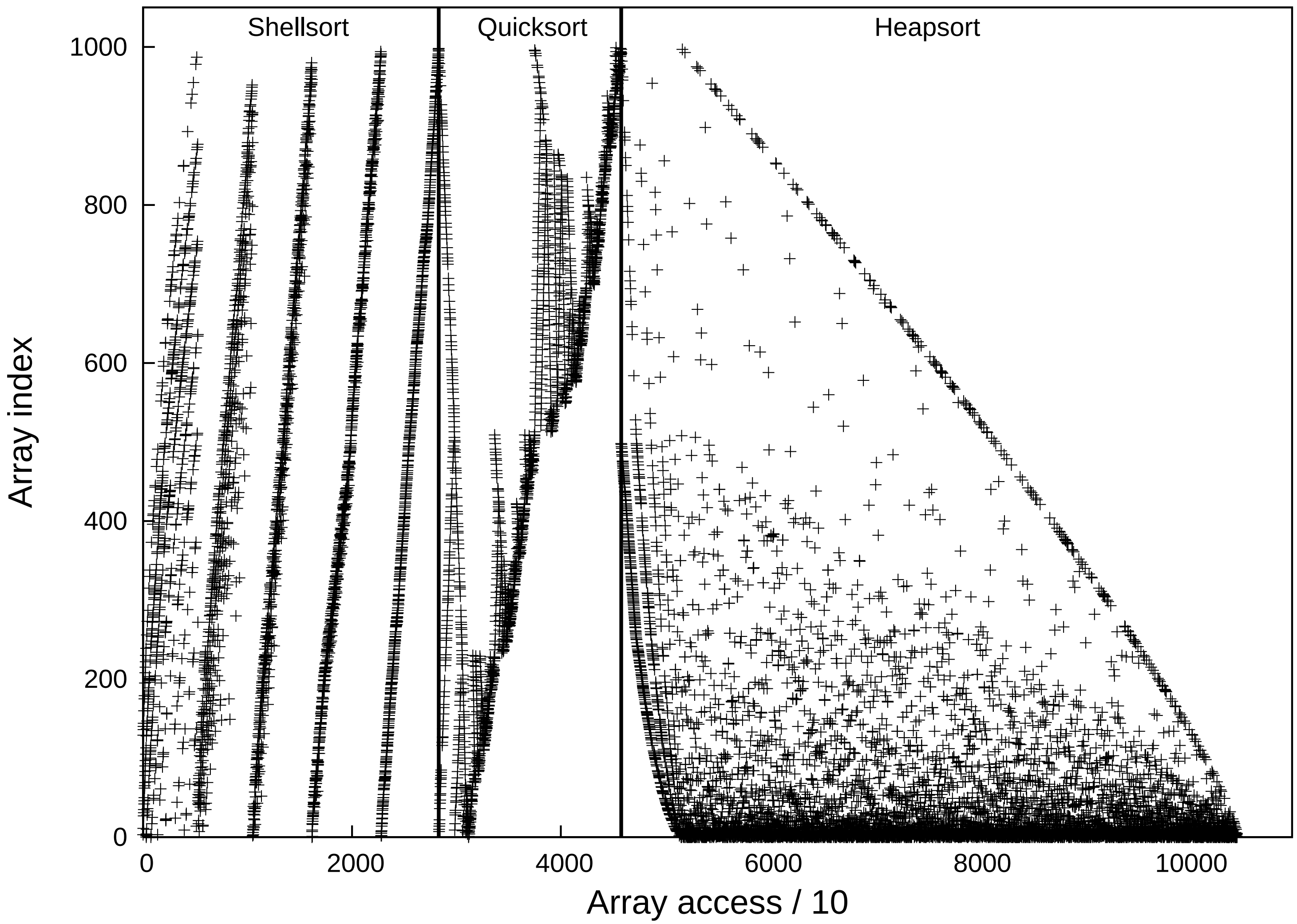
The C implementations are provided on the course materials webpage.

Method	Ordered	Reversed	Random	Random 0..9
Insertion	0.000s	7.903s	4.058s	3.524s
Selection	7.399s	8.091s	7.236s	7.466s
Bubble	15.699s	50.054s	54.328s	51.594s
Shell	0.005s	0.007s	0.027s	0.010s
Quick	5.985s	6.702s	0.016s	0.010s
Merge	0.023s	0.029s	0.037s	0.033s
Heap	0.027s	0.025s	0.034s	0.024s

Execution traces: insertion and selection sorts



Execution traces: shell, quick and heap sorts



Lower bound for comparison-based sorts

We've seen a range of comparison-based sorting methods, the best of which have $\Theta(N \log N)$ running time.

- Is it possible to do better (for any input)?

Consider the number of comparisons required to sort the integers 1 to N:

- There are $N!$ possible input permutations
- A sort must perform enough comparisons to uniquely identify any input permutation, else there are two permutations that are sorted the same
- An optimal binary comparison halves the search space in the worst case
  - Places candidate permutations in two disjoint categories. In the worst case the result of the comparison selects the larger category.
- Therefore, even if the comparisons are optimally chosen, we require at least $\log_2 N!$ comparisons to identify at least some input permutations

Lower bound for comparison-based sorts

How does $\log_2 N!$ grow? There are a few methods of analysis, including use of Stirling's approximation for the factorial function, but we use an easy-to-remember method here:

$$\begin{aligned}
\log_2 N! &= \log_2(1 \cdot 2 \cdot \ldots \cdot (N-1) \cdot N) \\
&= \log_2 1 + \log_2 2 + \ldots + \log_2(N-1) + \log_2 N \\
&= \sum_{k=1}^{N} \log_2 k \\
&= \Theta(N \log N)
\end{aligned}$$

We can make the final leap above because **we have seen this function before!**

- It's the cost function for adding N items to an empty heap.

Counting sort

If we know something about the range of keys then we can do better than $\Theta(N \log N)$.

- Counting sort assumes all keys are in the range 0 to $k-1$.
- It then counts the number of times each key appears in the input.

```
void counting_sort(int[] a, int[] b, int k, int N) {
    int[] c = new int[k];
    for ( i = 0; i < N; i++ )
        c[a[i]]++;
    for ( i = 1; i < k; i++ )
        c[i] += c[i-1];
    for ( i = N-1; i >=0; i-- ) {
        output_pos = --c[a[i]];
        b[output_pos] = a[i];
    }
}
```

Array $a[N]$ stores the unsorted input
Array $b[N]$ stores the sorted output
Array $c[k]$ stores the counts

Counting sort: example

- Step 1: create a cumulative count of input keys
- Step 2: work *backwards* through the input array, placing items in their final **stable** sorted position

Input array (a)

2	1	3	0	1	3	1

Count array (c)

1	3	1	2

	Output array (b)							Count array (c)			
1.								1	4	5	7
2.				1				1	3	5	7
3.				1			3	1	3	5	6
4.			1	1			3	1	2	5	6
5.	0		1	1			3	0	2	5	6
6.	0		1	1		3	3	0	2	5	5
7.	0	1	1	1		3	3	0	1	5	5
8.	0	1	1	1	2	3	3	0	1	4	5

Counting sort: analysis

How many times is each loop executed?

- N steps to scan the input array and count occurrences of each key.
- k steps to scan the count array and accumulate counts.
- N steps to produce the output array.

Thus counting sort has time (and space) cost $\Theta(N + k)$ in all situations.

- In most cases $k = O(N)$ and thus the counting sort is $\Theta(N)$.
- Typically used to sort a large array with only a small range of key values.

Counting sort is **stable**. This makes it useful for the next sort we'll look at...

Radix sort

Based on a sequence of *stable* sub-sorts on *sections* of the key space

- Radix sort starts with least-significant end of the key values
- ...and works towards the most-significant end
- Here we will assume we sort on one decimal digit at a time

Let's assume we are sorting d-digit key values. Then radix sort will perform a sequence of d sorts using a stable sorting algorithm.

329		72**0**		7**2**0		**3**29
457		35**5**		3**2**9		**3**55
657		43**6**		4**3**6		**4**36
839	$\longrightarrow$	45**7**	$\longrightarrow$	8**3**9	$\longrightarrow$	**4**57
436		65**7**		3**5**5		**6**57
720		32**9**		4**5**7		**7**20
355		83**9**		6**5**7		**8**39

Radix sort: analysis

Assuming the range of key space on each iteration is reasonably small, it makes sense to use *counting sort*.

- Thus the cost of each sub-sort is $\Theta(N + k)$, where k is the 'radix' (or base) of the radix sort.
- The overall running time is $\Theta(d(N + k))$ if the keys are d digits long when written in base k. The space cost is $\Theta(N + k)$.
  - Compare with time and space cost $\Theta(N + k^d)$ for an 'equivalent' single-pass counting sort. Exponential in d!

Assuming fixed-width key values, d and k are constants then we have cost (in time and space) of $\Theta(N)$.

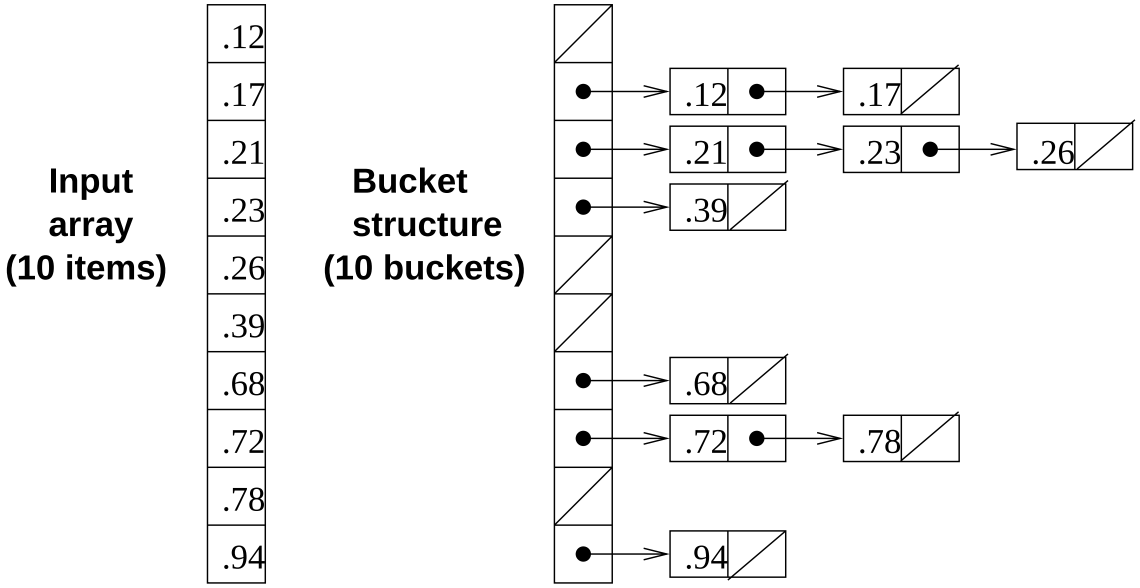
- But remember the hidden constant of proportionality may make this *slower* than comparison-based sorts in many situations

Bucket sort

Assume we know that key values are uniformly distributed over some range

- For example, the real numbers in the interval $[0, 1)$

Divide the interval into N equal-sized subintervals (or *buckets*). Represent each bucket as a linked list of input keys that fall in that subinterval, maintained in sorted order.



Bucket sort: analysis

Each input item is inserted into a list of, on average, $\Theta(1)$ items.

- This is provable if we assume each input key is a uniform r.v.
- So insertion in sorted order takes $\Theta(1)$ time on average
- ...but $\Theta(N)$ in the worst case

The second phase is to scan the list of N buckets, reading the items in sequential order and writing them back into the input array.

- This phase is always $\Theta(N)$

Overall, bucket sort is $\Theta(N)$ on average, and $\Theta(N^2)$ in the worst case.

- The first phase (insertion) dominates the running time

Order statistics

The kth *order statistic* of a set is the kth smallest element. How quickly can we find the kth order statistic, for any k?

- Upper bound $\Theta(N \log N)$: sort the list and return the kth value
- Potentially $\Theta(N)$ if you know something about the input distribution

It is easier to do better than $\Theta(N \log N)$ on average for any input set

- Use quicksort algorithm, but pick a random item as pivot and recurse only on one sub-partition

```
int select(int[] a, int l, int r, int k) {
    m = partition_exchange(a, l, r);
    if ( m == k ) return a[m];
    if ( m > k )  return select(a, l, m-1, k);
    else          return select(a, m+1, r, k);
}
```

Order statistics: analysis

The performance of the quicksort-based algorithm depends on good choice of pivot values.

Assuming the pivots are not too bad:

- Running time is $f(N) = f(N/2) + cN = \Theta(N)$
- Space complexity is $\Theta(\log N)$

But in the worst case:

- Running time is $f(N) = f(N-1) + cN = \Theta(N^2)$
- Space complexity is $\Theta(N)$

Order statistics: worst-case linear time

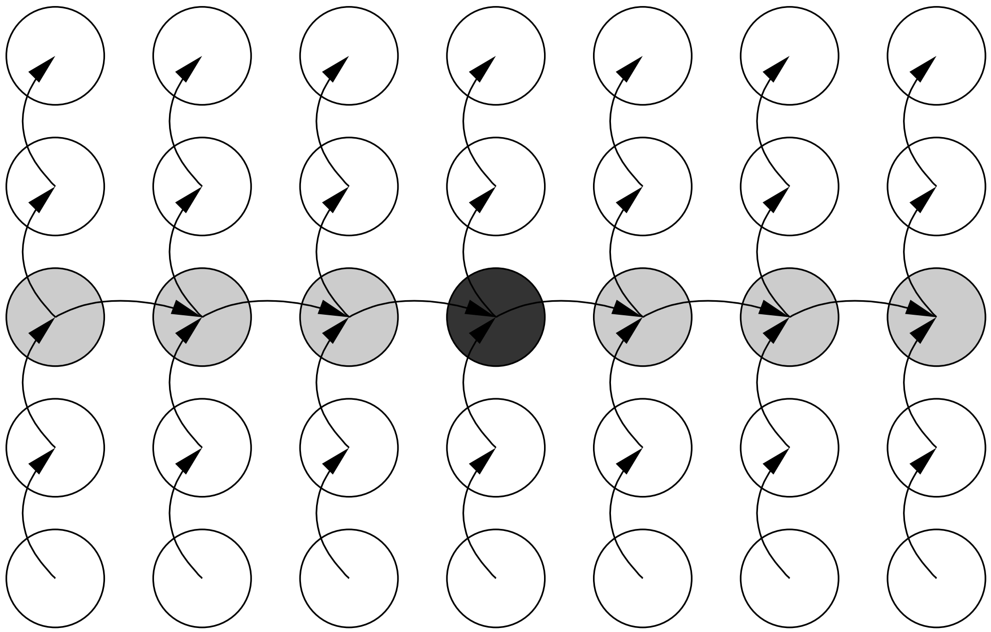
We can guarantee $\Theta(N)$ running time if we can be sure to always pick a good enough pivot. To do this, we:

- Divide the array into groups of five
- Store the median of each group of five in a temporary array
- Recursively call `select()` to find the median of medians-of-fives
- Use the median of medians-of-fives as the pivot value

```
int select(int[] a, int l, int r, int k) {
    int[] b = medians_of_fives(a, l, r);
    pivot = select(b, 0, b.len-1, b.len/2);
    m = partition_exchange(a, l, r, pivot);
    if ( m == k ) return a[m];
    if ( m > k )  return select(a, l, m-1, k);
    else          return select(a, m+1, r, k);
}
```

Median of medians-of-five

Why is the median of medians-of-fives (M) good enough?



Known to be less than M:
$N/10$ of the medians-of-fives
2 further values per median-of-five

$3N/10$ values in total

Known to be greater than M:
$N/10$ of the medians-of-fives
2 further values per median-of-five

$3N/10$ values in total

Thus we *know* that, if we use M as pivot, the largest sub-partition will contain at most $7N/10$ values.

What about duplicate keys?

The explanation about why the meadian-of-medians M is 'good enough' assumed that all input keys are unique.

- In fact we can easily extend the reasoning to handle duplicate keys
- Duplicate keys simply mean that:
  - ...we have $3N/10$ items less than **or equal to** M
  - ...and $3N/10$ items greater than **or equal to** M
- The only thing we need to handle is the possibility that the input contains lots of duplicates of M
- Fortunately there are variants of `partition_exchange()` which specifically deal with duplicates of the pivot
  - Run in $\Theta(N)$ time. Neither partition contains any key $= M$.
  - Still the case that neither partition contains more than $7N/10$ items

Worst-case linear time: cost analysis

Two recursive invocations of `select()`, plus linear-time work to find the median of medians-of-fives, and to perform partition-exchange around the chosen pivot: $f(N) = f(N/5) + f(7N/10) + \Theta(N)$.

If $f(N) = O(N)$ then we can show that $\forall N > N_0 \, . \, f(N) \leq cN$:

$$\begin{aligned} f(N) &= f(N/5) + f(7N/10) + \Theta(N) \\ &\leq cN/5 + 7cN/10 + \Theta(N) \\ &\leq \tfrac{9}{10}cN + dN \\ &\leq cN \quad \text{(for any } c \geq 10d\text{)} \end{aligned}$$

We proved the asymptotic bound by substiting cN and showing there exists some suitable c. **You cannot simply subsitute a $O()$ or $\Theta()$ term!**

We must also establish the base case for some N_0. This is of course trivial, as we can set c as big as we like (we need only prove $f(N_0) \leq cN$)!

Worst-case linear time: practical performance

In practise the worst-case linear time algorithm has a very high constant of proportionality which will in general put it beyond practical use unless you absolutely must have guaranteed linear growth.

In part this is due to the size of the recursive sub-problems:

- $\frac{7}{10}N + \frac{1}{5}N \; = \; \frac{9}{10}N$
- Recall that $f(N) \; = \; f(\alpha N) + N \; = \; \frac{1}{1-\alpha}N$
- As α approaches 1, the constant of proportionality approaches ∞
- $\alpha = \frac{9}{10}$ is closer to 1 than we would like

However, the sheer number of array accesses and comparisons needed to find an appropriate pivot mean that the $\Theta(N)$ term in the recurrence formula also hides a large constant of proportionality.

Cost functions: asymptotic approximations

Consider binary search in a sorted array. The number of array elements accessed in the worst case can be written at a number of levels of detail:

$f(N) = \lfloor \log_2 N \rfloor + 1$	Accurate for all N
$f(N) \approx \log_2 N$	Asymptotically accurate (relative error tends to zero)
$f(N) = \Theta(\log N)$	Asymptotically bounded (relative error is an unknown constant factor)

- Pick the simplest representation that provides sufficient information
- Often an asymptotic bound is good enough to classify an algorithm
- Sometimes you care about the constant factor
  - Then an asymptotically accurate formula will usually suffice
  - You will *almost never* care about the really fine detail!

Proving asymptotic bounds for recurrence formulae

Consider $f(N) \;=\; 2f(N/2) + \Theta(N)$

- Clearly any solution can only be a bound, due to the big-theta term
- It's tempting **but wrong** to substitute a big-theta solution:
  - Try $f(N) \;=\; \Theta(N)$: $f(N) \;=\; 2\Theta(N/2) + \Theta(N) \;=\; \Theta(N)$

The correct solution is to prove *one* bound by substituting an inequality.

- Try $f(N) \;\leq\; cN$:
  - $f(N) \;\leq\; 2(cN/2) + \Theta(N) \;\leq\; (c+d)N \;\not\leq\; cN$ ✘
- Try $f(N) \;\leq\; cN\log_2 N$:

$$\begin{aligned} f(N) &\leq 2(c(N/2)\log_2(N/2)) + \Theta(N) \\ &\leq cN(\log_2 N - 1) + dN \\ &= cN\log_2 N + (d-c)N \\ &\leq cN\log_2 N \quad \text{(if } c \geq d\text{)} \quad ✔ \end{aligned}$$

Searching

- Searching is our second big set-piece topic
- Retrieval of information from previously stored data records
- We can present the problem as a set ADT

`s = empty_set()`	returns a new, empty set
`insert(s, k, x)`	insert key `k` and associated data `x` into set `s`
`remove(s, k)`	remove key `k` from set `s`
`x = search(s, k)`	search for data associated with key `k` in set `s`
`x = select(s, i)`	find data associated with the `i`th largest key in `s`
`x[] = sort(s)`	return data items in sorted order of key value

- We will investigate various implementations of this ADT which efficiently support all (or most) of the operations listed above

Set ADT in Java

In the pseudo-Java examples, I'll assume the ADT is defined by an interface something like:

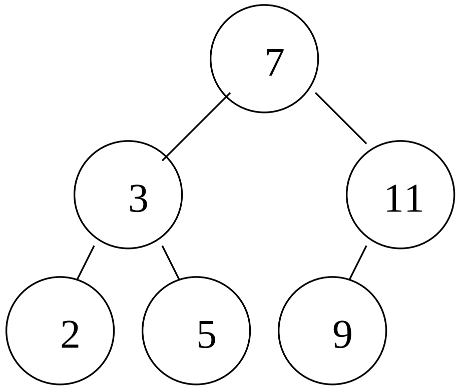
```
interface Set {
    void insert(Item x);
    void remove(int k);
    Item search(int k);
    Item select(int i);
    Item[] sort();
}
```

`Item` is a base class that can be extended by users of the `Set` interface. It is defined to contain a search key for use by the ADT.

```
class Item {
    int key;
}
```

Binary search tree

- A binary tree in which each node stores a data item
  - Each node's key value is greater than any key in its left subtree
  - ...and less than any key in its right subtree




```
class Node {
    Item item; /* data item and key value */
    Node l, r; /* left, right subtrees */
}
```

- Usually implemented as dynamically-allocated node structures which contain pointers to their left and right subtrees

Searching in a BST

- Compare search key k_s with root node's key k_r
  - $k_s = k_r$? Then we're done
  - $k_s < k_r$? Then recurse on the left subtree
  - $k_s > k_r$? Then recurse on the right subtree

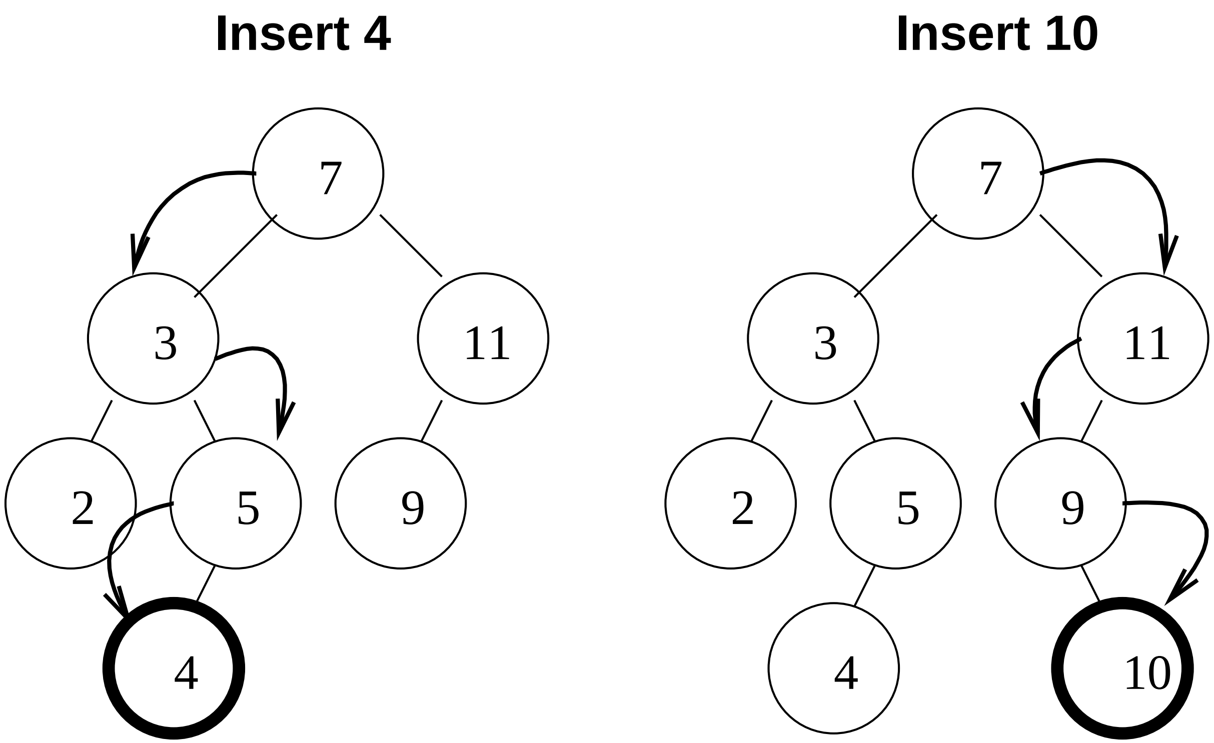
```
Item search(Node root, int k) {
    if ( root == null )
        return null;                /* failed */
    if ( k == root.item.key )
        return root.item;           /* succeeded */
    if ( k < root.item.key )
        return search(root.l, k);   /* recurse left */
    else
        return search(root.r, k);   /* recurse right */
}
```

Insertion in a BST

- Like search, but:
  - If we find a match, replace the data item
  - If we do not find a match, insert a new leaf node

```
void insert(Node root, Item new)) {
    Node parent, curr = root;
    while ( (curr != null) && (new.key != curr.item.key) ) {
        parent = curr;
        if ( new.key < curr.item.key ) curr = curr.l;
        else                           curr = curr.r;
    }
    if ( curr == null ) { /* no existing node with this key value? */
        curr = new Node();
        if ( new.key < parent.item.key ) parent.l = curr;
        else                             parent.r = curr;
    }
    curr.item = new; /* insert the supplied data item */
}
```

Insertion in a BST: example



BST: cost analysis

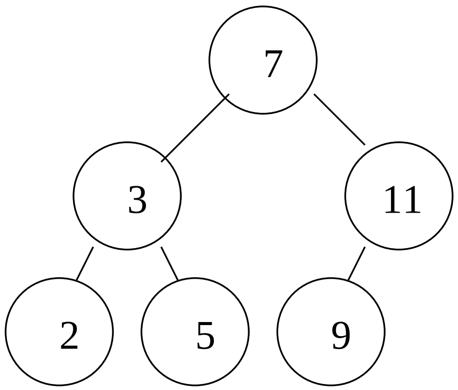
How long does it take to search or insert in a BST?

- Depends on the number of nodes traversed
- Longest path from root node to a leaf node
- If the tree is *optimal* then the longest and shortest paths differ by at most one
  - In which case approx $\log_2 N$ nodes are traversed in the worst case
- But a really unbalanced tree could require up to all N nodes to be traversed (think about every node having no more than one subtree)
- How balanced is a *random* BST likely to be?
  - The metrics we are interested in are the *total path length* and the *average path length*

BST: Average path length (1)

The path length of a node is the number of 'hops' it lives from the root

- The *total path length* (also known as *internal path length*) is the sum of the path lengths of all nodes
- The *average path length* is simply the total path length divided by N



In the example above, the internal path length is $(3 \cdot 2 + 2 \cdot 1 + 0) = 8$

- The *average path length* is $8/6 \approx 1.33$

BST: Average path length (2)

Let C_N be the total path length of a random BST.

$$\begin{aligned} C_0 &= C_1 = 0 \\ C_N &= N - 1 + \tfrac{1}{N}\textstyle\sum_{k=1}^{N}(C_{k-1} + C_{N-k}) \\ &= N - 1 + \tfrac{2}{N}\textstyle\sum_{k=1}^{N} C_{k-1} \end{aligned}$$

Algebraic manipulation to get rid of the summation:

$$\begin{aligned} NC_N &= N(N-1) + 2\textstyle\sum_{k=1}^{N} C_{k-1} \\ (N-1)C_{N-1} &= (N-1)(N-2) + 2\textstyle\sum_{k=1}^{N-1} C_{k-1} \\ NC_N - (N-1)C_{N-1} &= N(N-1) - (N-1)(N-2) + 2C_{N-1} \end{aligned}$$

BST: Average path length (3)

Further simplification:

$$
\begin{aligned}
NC_N - (N-1)C_{N-1} &= N(N-1) - (N-1)(N-2) + 2C_{N-1} \\
NC_N &= (N+1)C_{N-1} + (N^2 - N) - (N^2 - 3N + 2) \\
&= (N+1)C_{N-1} + 2N - 2
\end{aligned}
$$

We want a recursive term on the RHS that is in the same form as the LHS. To obtain this, divide through by $N(N+1)$:

$$
\begin{aligned}
\tfrac{C_N}{N+1} &= \tfrac{C_{N-1}}{N} + \tfrac{2}{N+1} - \tfrac{2}{N(N+1)} \\
&= \tfrac{C_1}{2} + (\tfrac{2}{3} - \tfrac{2}{12}) + \ldots + (\tfrac{2}{N} - \tfrac{2}{(N-1)N}) + (\tfrac{2}{N+1} - \tfrac{2}{N(N+1)}) \\
&= \textstyle\sum_{k=1}^{N} (\tfrac{2}{k+1} - \tfrac{2}{k(k+1)})
\end{aligned}
$$

BST: Average path length (4)

Now we invent some approximations to the RHS which allow further simplification:

$$\begin{aligned}\frac{C_N}{N+1} &= 2\sum_{k=1}^{N}\left(\frac{1}{k+1}-\frac{1}{k(k+1)}\right)\\ &\approx 2\sum_{k=1}^{N}\left(\frac{1}{k}-\frac{1}{k^2}\right)\end{aligned}$$

This can be approximated with a definite integral:

$$\begin{aligned}\frac{C_N}{N+1} &\approx 2\int_1^N\left(\frac{1}{x}-\frac{1}{x^2}\right)dx\\ &= 2\left[\ln x+\frac{1}{x}\right]_1^N = 2(\ln N+\frac{1}{N}-1)\approx 2\ln N\end{aligned}$$

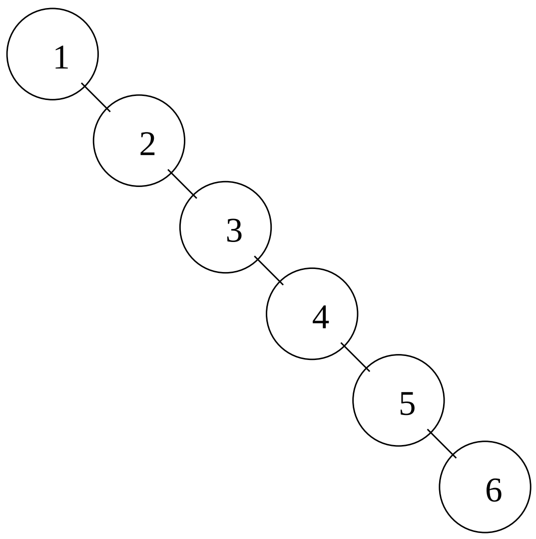
Thus $C_N \approx 2(N+1)\ln N \approx 2N\ln N \approx 1.39N\log_2 N$ and the average path length is approximately $1.39\log_2 N$.

- A search of a random BST visits about 40% more nodes than in an optimum BST.

BST: worst-case cost

In the worst case we can end up with a very unbalanced BST.

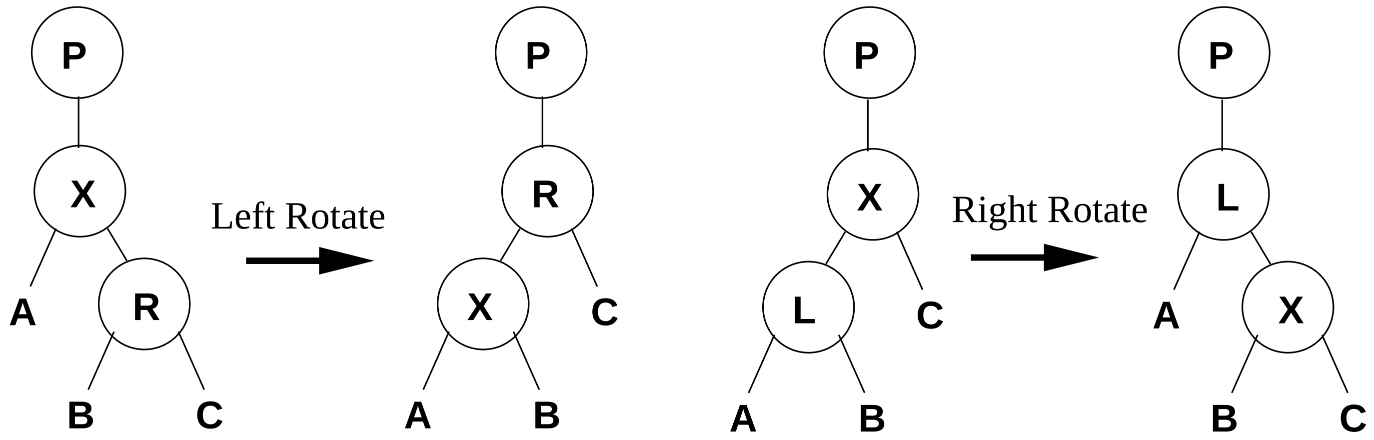
- As with quicksort, this rather inconveniently happens when the input data is already ordered!
- Hence the worst case is worryingly likely to occur in practice



BST: restructuring and rotations

It is possible to avoid worst-case performance if we periodically *restructure* the BST to avoid it becoming too unbalanced. Although many restructuring strategies have been developed, they are almost all based on the same simple primitive: **rotation**

- A rotation is a local tree transformation which affects only two nodes and three links. Two symmetric variants: left and right rotation.



- Careful and repeated application of this efficient transformation can ensure a BST remains reasonably balanced

BST: implementing rotation

Implementating rotation is straightforward:

```
Node rotate_left(Node x) {
    Node r = x.r;
    x.r = r.l;
    r.l = x;
    return r;
}
```

```
Node rotate_right(Node x) {
    Node l = x.l;
    x.l = l.r;
    l.r = x;
    return l;
}
```

Note that these routines do not update the parent's reference.

- Because they cannot find the parent! (And what if x is root?)
- Often we add a parent reference allowing the tree to be traversed in both directions (like a double linked list)

Instead they return the new root of the subtree previously rooted at x.

- The caller must ensure the parent is updated appropriately.

BST: brute-force rebalancing (1)

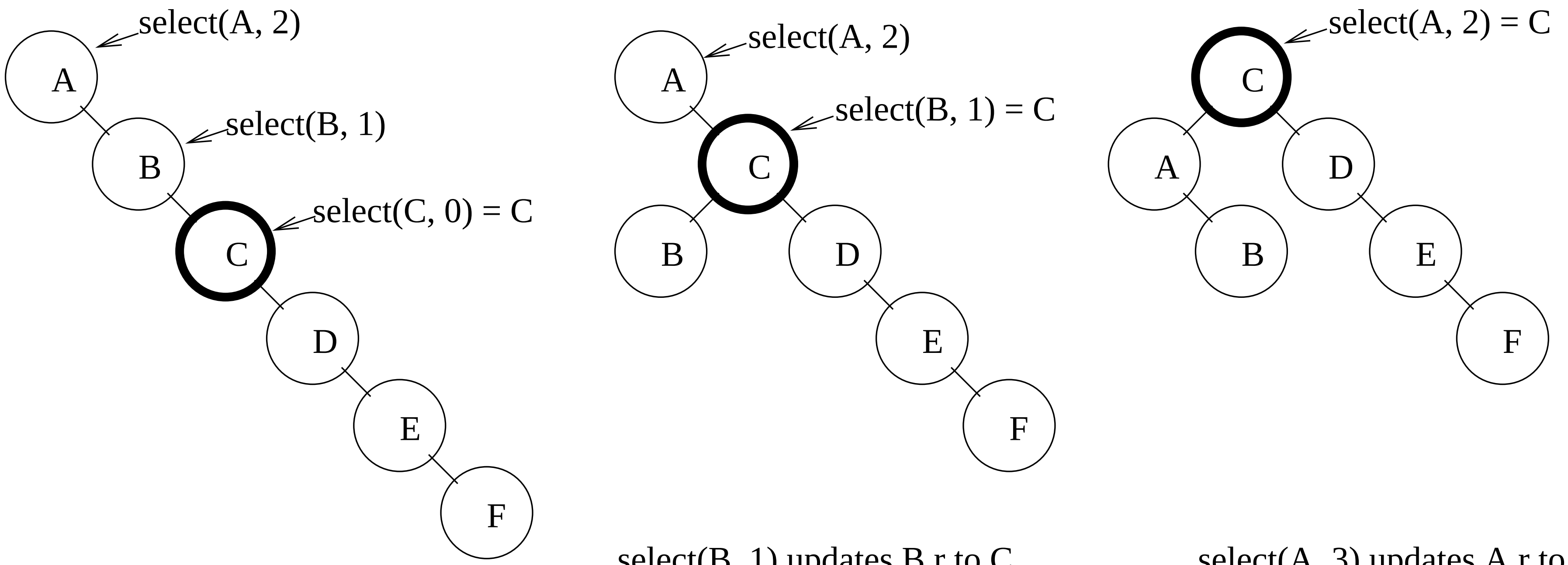
First we need a simple function to count the number of nodes in a subtree:

```
int count(Node root) {
    if ( root == null ) return 0;
    return 1 + count(root.l) + count(root.r);
}
```

We can use this to implement a move-kth-to-root selection algorithm:

```
Node select(Node root, int k) {
    int rk = count(root.l);
    if ( rk > k) {
        root.l = select(root.l, k); /* root.l is k'th item */
        root = rotate_right(root);  /* root   is k'th item */
    } else if ( rk < k) {
        root.r = select(root.r, k - rk - 1);
        root = rotate_left(root);
    }
    return root; /* k'th item is at root of tree */
}
```

BST: brute-force balancing (2)



select(C, 0) returns C

select(B, 1) updates B.r to C and rotates left at B, returning new root node (C)

select(A, 3) updates A.r to C and rotates left at A, returning new root node (C)

BST: brute-force balancing (3)

With the trickery engineering hidden in `select`, the balancing routine itself is quite straightforward:

```
Node balance(Node root) {
    int N = count(root);          /* N == number of items     */
    root = select(root, N/2);     /* move median key to root */
    root.l = balance(root.l);     /* balance left subtree     */
    root.r = balance(root.r);     /* balance right subtree    */
    return root;                  /* inform caller of new root */
}
```

- Moves median item to root of subtree
- Then recursively does the same for each subtree
  - Each of which now contains approx. $N/2$ items as required

BST: deletion (1)

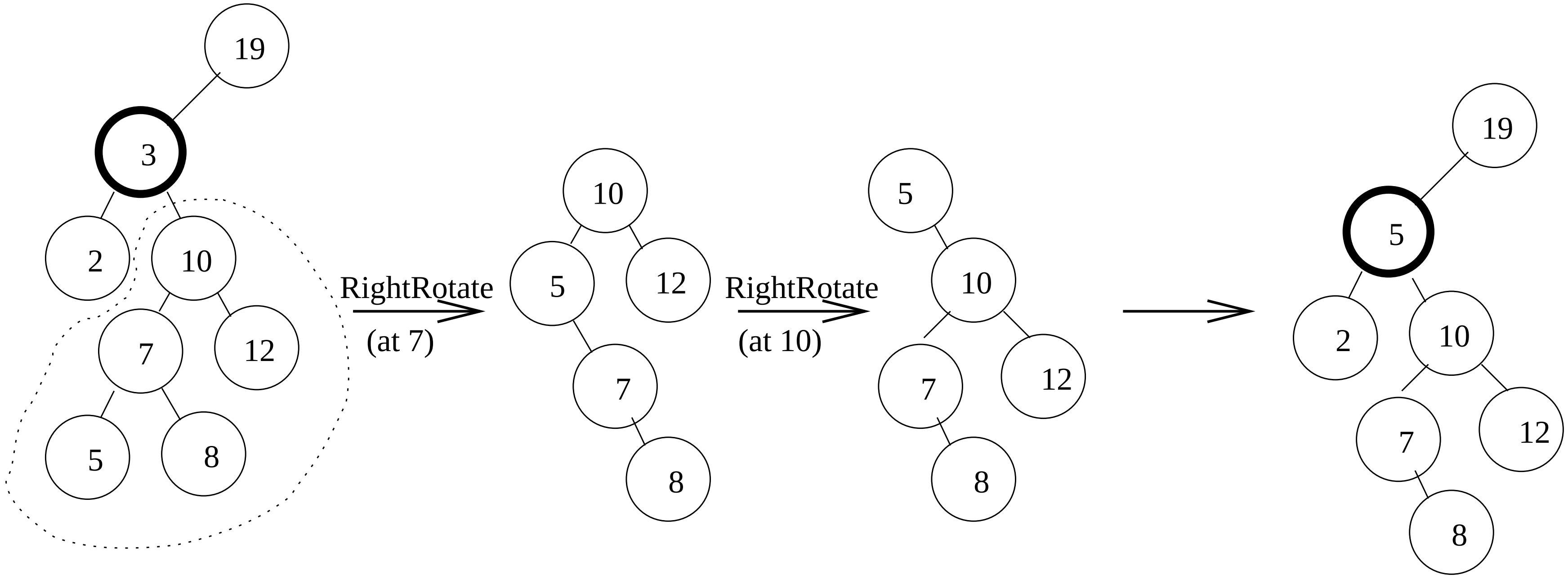
Deletion is straightforward unless the node has two non-empty subtrees

- No subtrees: update parent link to `null`
- One subtree: update parent link to that subtree

In the general case we need some way of joining together the two subtrees of the deleted node.

- **Trick:** move the smallest node in the right subtree to root
  - For example, use the move-kth-to-root algorithm, $k = 0$
  - That node will have no left subtree (since it is smallest)
  - So attach the deleted node's left subtree

BST: deletion (2)



Note that moving the smallest node to be new root can be achieved *iteratively* and *without rotations*.

- The approach here is to demonstrate the concept
- A real implementation would perform more special-cased tree walking and pointer manipulations

BST: deletion (3)

Recursive implementation returning new root of subtree allows us to update the parent link:

```
Node delete(Node root, int k) {
    if ( k < root.item.key ) {
        root.l = delete(root.l, k);    /* recurse left */
    } else if ( k > root.item.key ) {
        root.r = delete(root.r, k);    /* recurse right */
    else if ( root.r == null ) {
        root = root.l;                 /* 0 or 1 subtrees */
    } else {
        Node x = select(root.r, 0);    /* 1 or 2 subtrees */
        x.l = root.l;
        root = x;
    }
    return root;
}
```

BST: traversing all nodes

A common BST operation is visiting every node in the tree in order.

- The recursive implementation is trivial
- An iterative implementation will require parent pointers!

```
int visit(Node root, Item[] sorted, int i) {
    if (root == null) return i;
    i = visit(root.l, sorted, i);    /* Visit left subtree */
    sorted[i++] = root.item;         /* Visit root */
    i = visit(root.r, sorted, i);    /* Visit right subtree */
    return i;
}
Item[] sort(Node root)) {
    Item[] sorted = new int[count(root)];
    visit(root, sort, 0);
    return sorted;
}
```

BST: cost summaries

Operation	Average time	Worst-case time
Search	$O(\log N)$	$O(N)$
Insert	$O(\log N)$	$O(N)$
Delete	$O(\log N)$	$O(N)$
Select	$O(N)$	$O(N)$
Sort	$O(N)$	$O(N)$

Space complexity of all operations is $O(\log N)$ average and $O(N)$ worst case, if implemented recursively. All these operations have iterative implementations!

Select is $O(N)$ even on average because of the need to count items in the left subtree

- `count()` can be implemented $O(1)$ if each node stores the number of items in its subtree
- But balance this against cost of maintaining that state

Splay trees (1)

BSTs have very good average performance, but an unfortunate worst case.

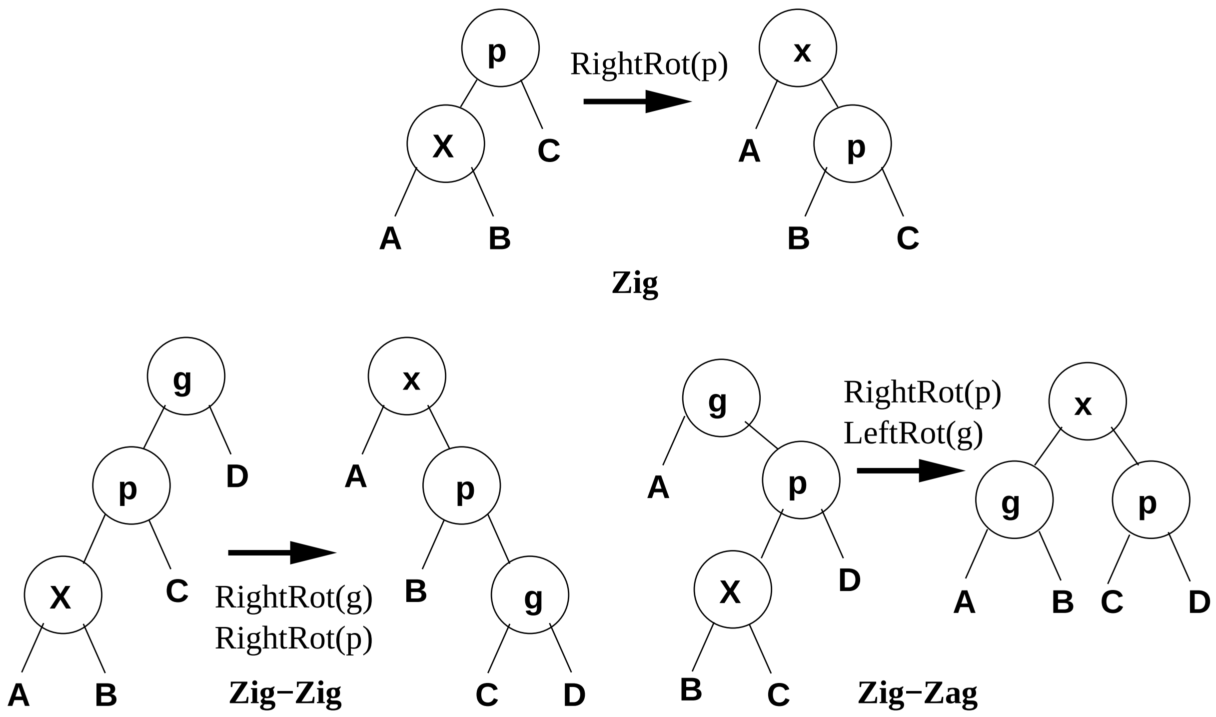
- Splay trees provide a good *amortised* worst-case bound on running time.
- Splay trees require no special representation. They are normal BSTs.
- They achieve their surprising properties with a rather simple move-to-root strategy for searches.

Consider the `select()` algorithm which moved the kth smallest value to the root of a BST.

- We could easily imagine modifying that algorithm to move-to-root the node that matches a search key
  - Simply a move-to-root search rather than a move-to-root select.
- ...Or modify it to insert and then move-to-root a new data item

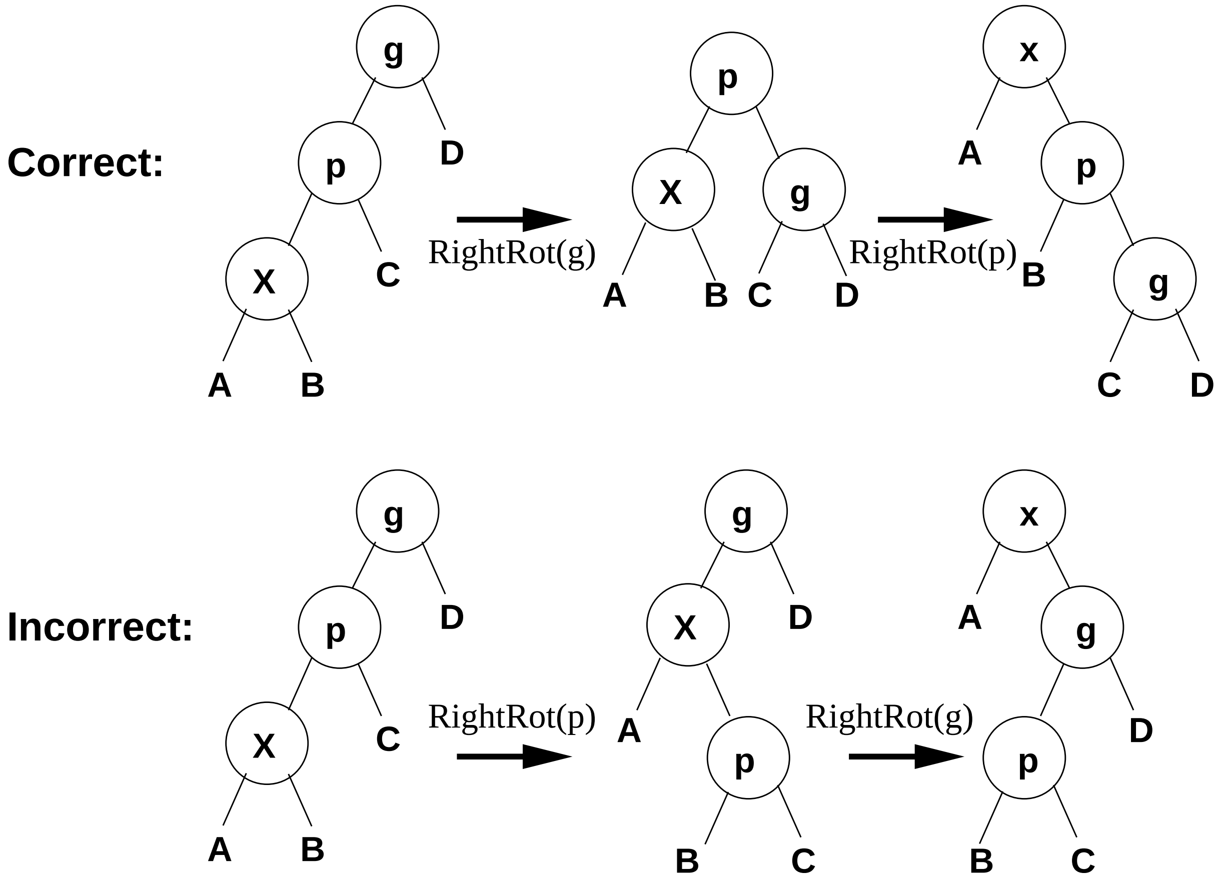
Splay trees (2)

The *only* twist when *splaying* a node (to move it to root) is that we consider double rotations rather than single rotations:



Splay trees: 'Zig-zig' splaying

Note that we effectively apply a sequence of single rotations at the parent of x, unless $x < p < g$ or $x > p > g$.



Splay trees: cost summary

The number of comparisons required for any sequence of M insert or search operations in an N-node splay tree is $O((N+M)\log(N+M))$

- The performance of any single operation could be bad: $O(N \log N)$
  - In fact you should be able to convince yourself the worst case is $O(N)$
- If $M = O(N)$ we have guaranteed average cost $O(\log N)$ per operation
  - Expensive operations are amortised across others which are *guaranteed* to be cheaper.

Furthermore, the cost of any sequence of splay operations is within a constant factor of accesses into *any* static tree.

- Even a purpose-built one with frequently-accessed nodes near the root.

For this course we state these results **without proof**

Splay trees: practical concerns

There are a number of reasons why you might not choose a splay tree for your own applications:

1. The asymptotic bounds hide a large constant of proportionality: moving every accessed item to the root is obviously a significant overhead.

2. You are only guaranteed a good asymptotic bound across a significant number of operations. Sometimes you will care about every operation being 'not too bad'.

On the other hand, splay trees are simple to understand (and to implement) and may be a good choice if you know that the distribution of key accesses is not uniformly random.

- A key feature of splay operations is that they adapt to non-uniform access patterns

- With a heavily-skewed distribution you can achieve better than $O(\log N)$ performance!

2-3-4 trees

BSTs have very good average-case performance, but an unfortunate worst case. Splay trees have a good *amortised* worst-case bound.

- 2-3-4 trees guarantee logarithmic running time of every basic operation
- But this is at the cost of a more complex representation.

The rules of 2-3-4 trees:

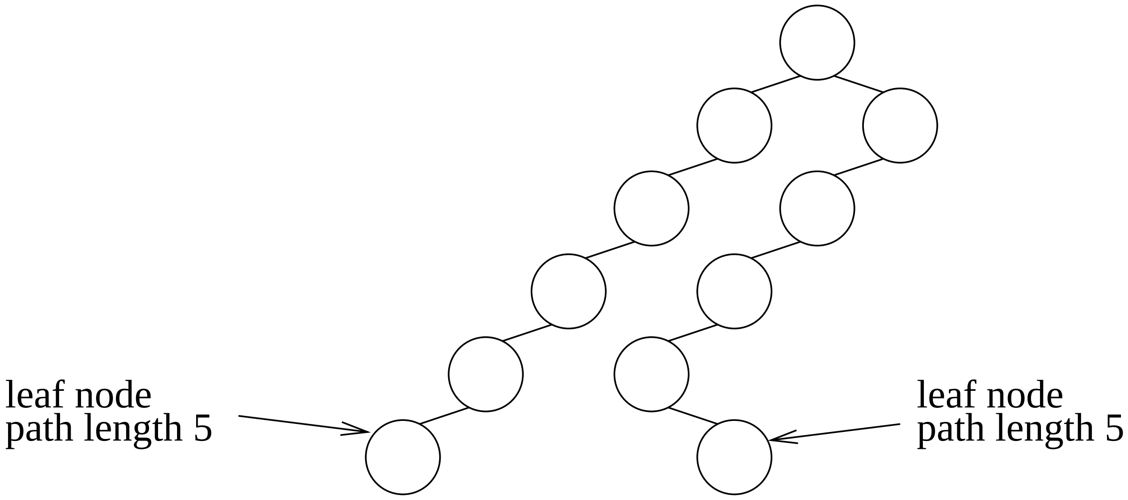
1. Each node stores 1, 2, or 3 data items (and key values).
2. Each node has 2, 3, or 4 child pointers.
3. All pointers to empty subtrees are the same distance from the root

- 2-3-4 trees are by definition **perfectly balanced!**

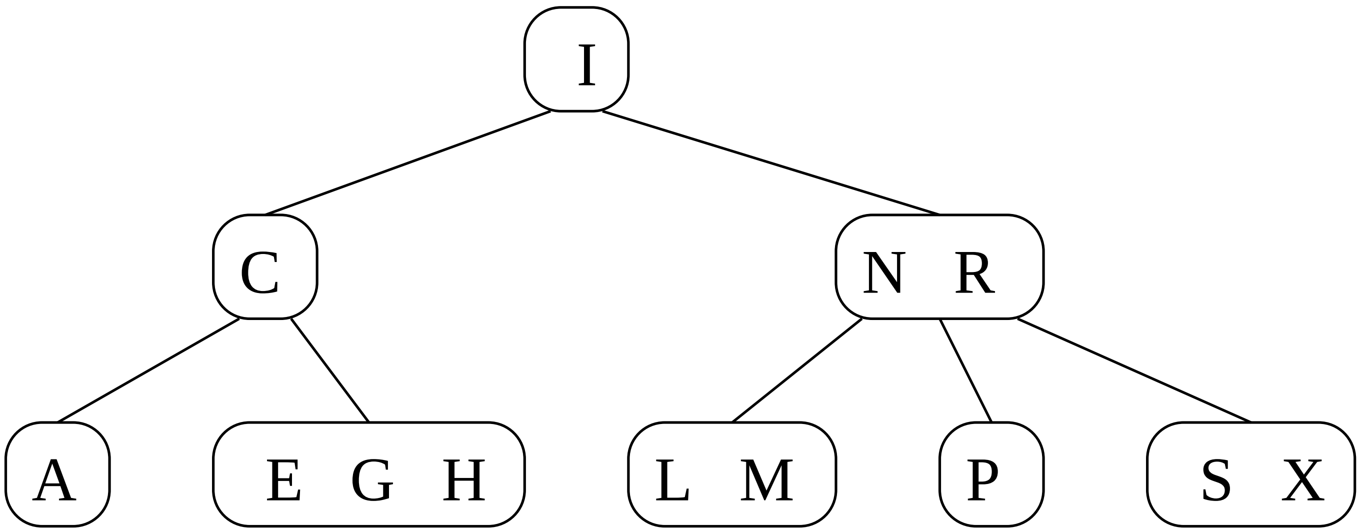
Explanation of third rule

All **pointers to empty subtrees** are the same distance from the root.

- This is not the same as saying all **leaf nodes** are the same distance from the root!
- The former statement is stronger, and guarantees optimal balance.
- The latter is satisfied by the following obviously unbalanced tree:



2-3-4 trees: example

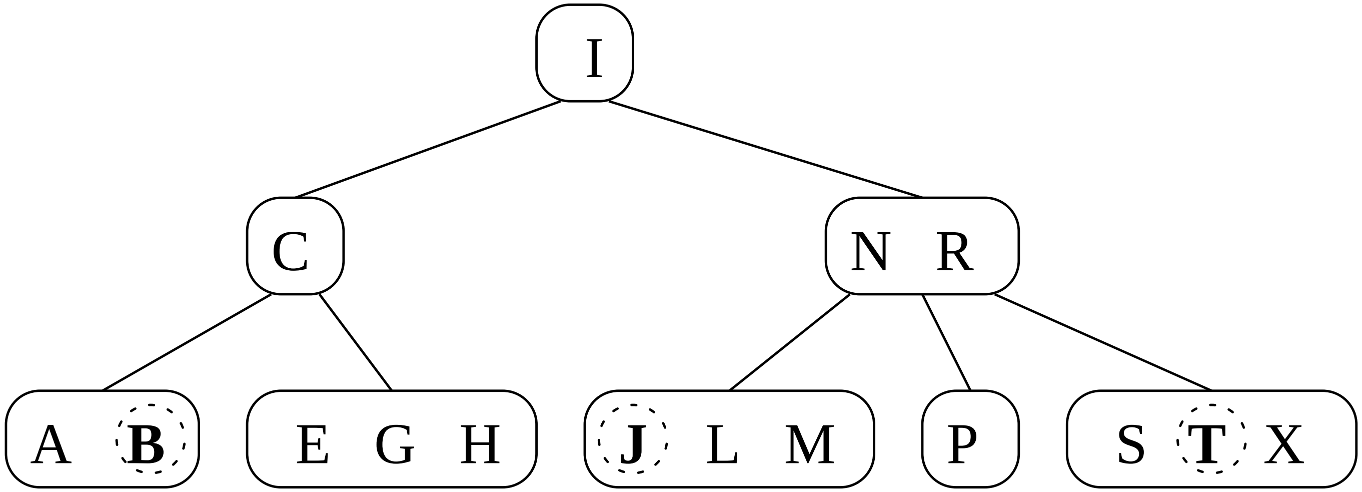


- Data values are maintained in order in 3 and 4 nodes
  - Divide the search space into 3 or 4 ordered partitions
- Search algorithm is slightly more complicated
  - Work at each node depends on the node size
- And how do we maintain the path-length invariant?

2-3-4 trees: insertion

Search until we reach a leaf node.

- Insert the new key value in sorted order in the leaf node.

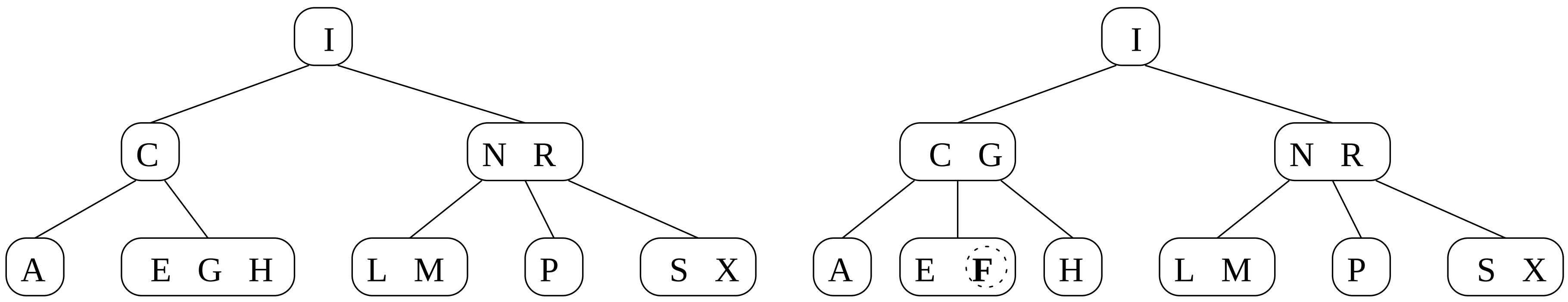


- How would we insert key value **F**?
  - Creating a new leaf node would break the path-length rule
  - So split the 4-node into two 2-nodes...

2-3-4 trees: splitting 4-nodes

The median key value is moved into the parent node.

- The new key value can then be inserted into the appropriate new 2-node (making it a 3-node).



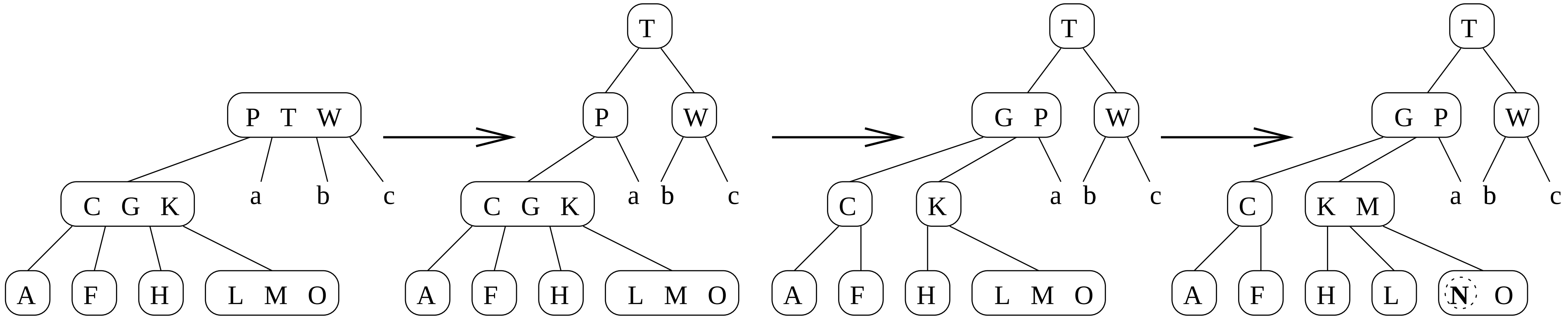
But what if the parent is a 4-node?

- And the grandparent? And so on?

2-3-4 trees: top-down splitting

We can avoid an avalanche of node splittings back up the tree by *pre-emptively* splitting any 4-nodes that we visit on the way down the tree

- Consider inserting **N** into the tree below

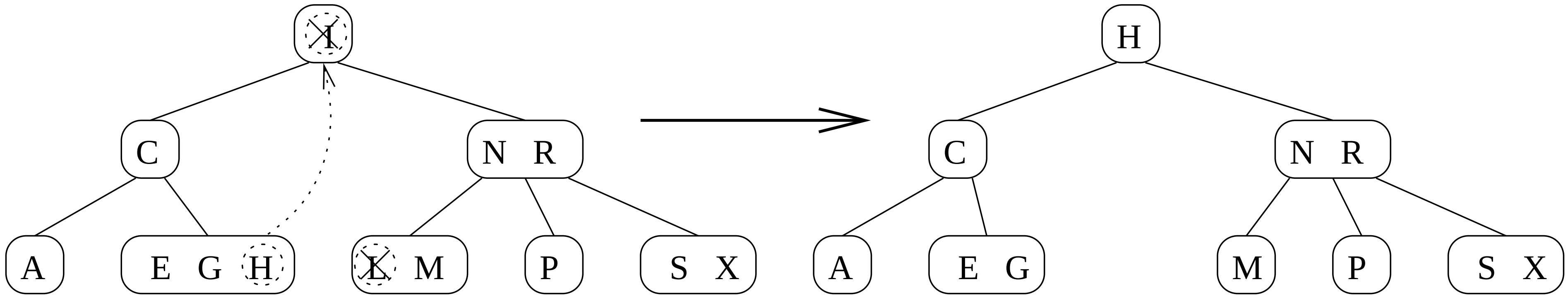


- Notice that splitting the root adds one to the path length of all other nodes in the tree
  - Tree height is only ever changed at the root of a 2-3-4 tree

2-3-4 trees: deletion

Find the node containing the key value to delete

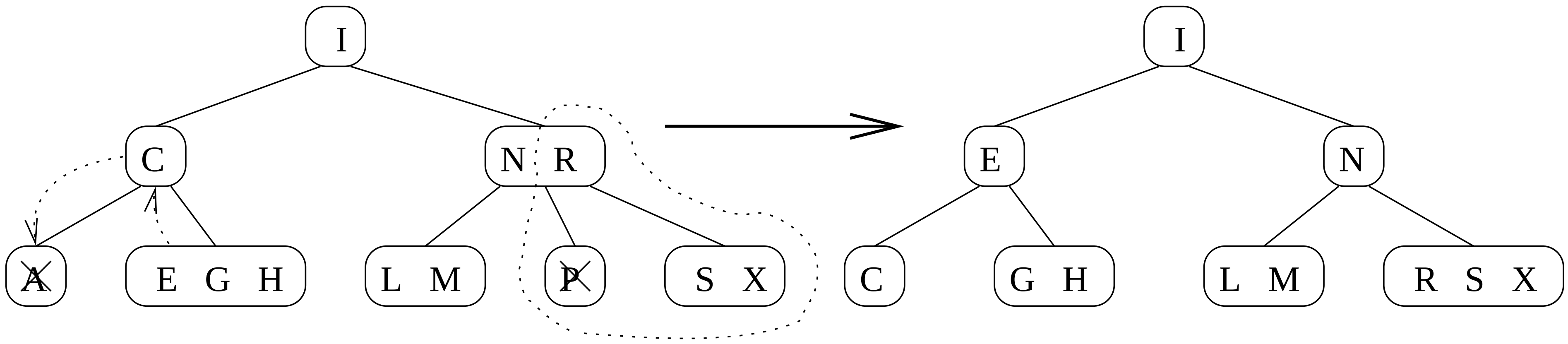
- Assume it is the ith key in its node
- If it's a leaf node, then remove the ith key
- Else find the largest key in the ith subtree and use it as replacement



2-3-4 trees: key transfers and node merging

What if you end up with an empty leaf node?

- If an adjacent sibling is a 4-node, transfer a key via the parent
- If an adjacent sibling is a 2- or 3-node, *merge* the two nodes and steal a key from the parent

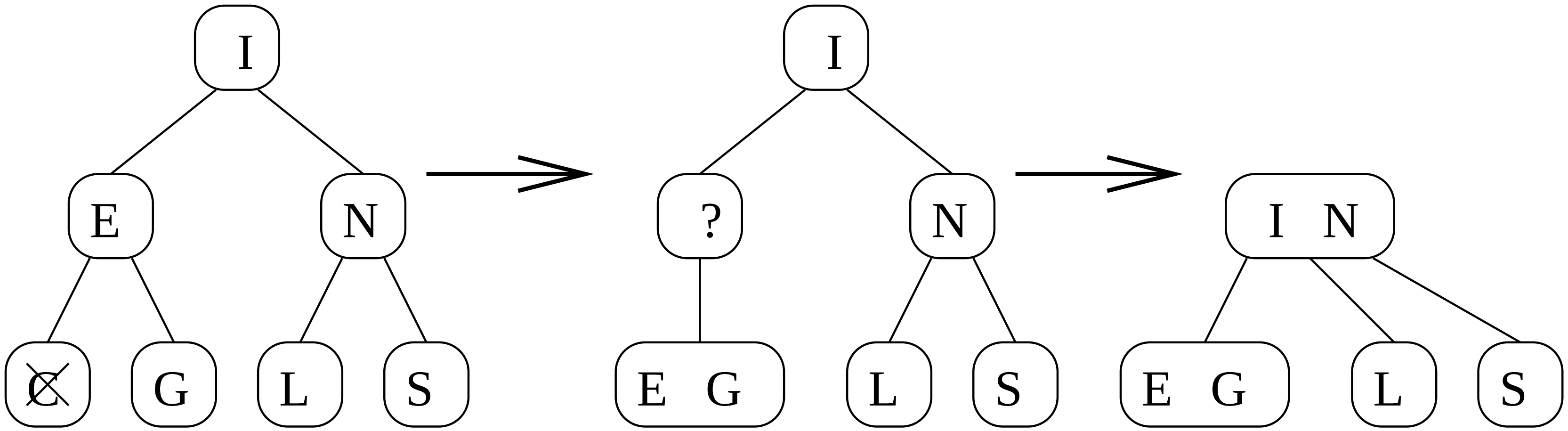


2-3-4 trees: cascading node merges (1)

When merging two siblings, what if the parent underflows?

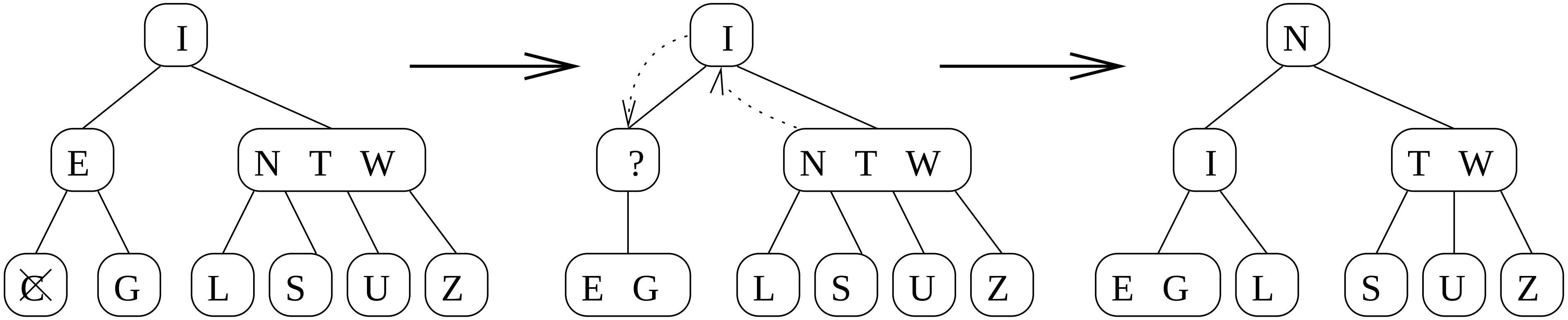
- Transfer a key from an adjacent 4-node sibling
- Merge with an adjacent 2- or 3-node sibling

This can propagate all the way to the root, reducing tree height by one



2-3-4 trees: cascading node merges (2)

Another example, with a transfer between non-leaf siblings:



- Note how one of the subtrees is also transferred

2-3-4 trees: node representation

```
class Node234 {
    int nr_items; /* 1, 2, or 3 */
    Item item1, item2, item3;
    Node child1, child2, child3, child4;
}
```

You should get the feeling that this representation is

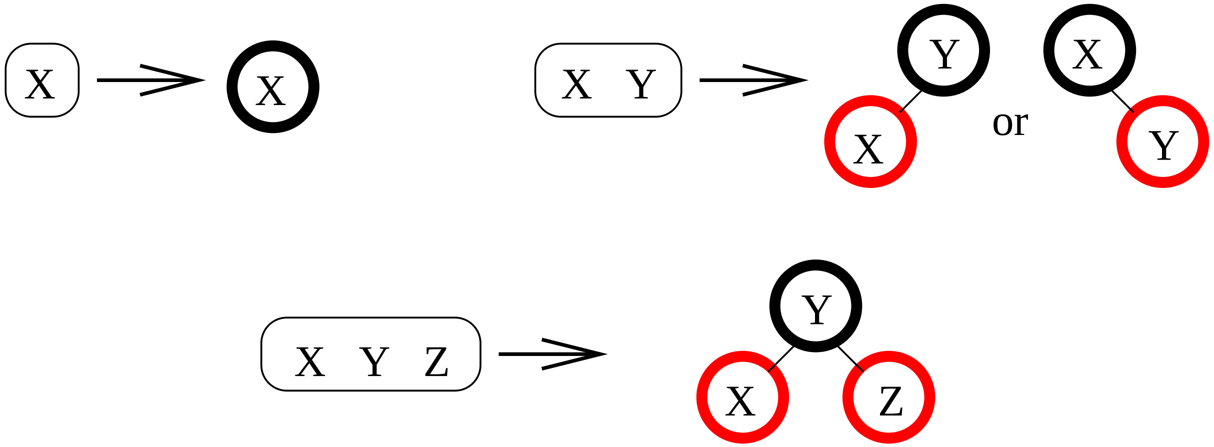
- Inefficient in space for 2 and 3 nodes
- A pain to implement operations for (multiple cases to handle)
  - Per-node algorithm complexity will affect the constant of proportionality of the cost functions

Can we get the guaranteed balance of 2-3-4 trees with the simple representation of binary trees?

Red-black trees

We can represent 2-3-4 trees as standard BSTs if we add one extra bit of information per node

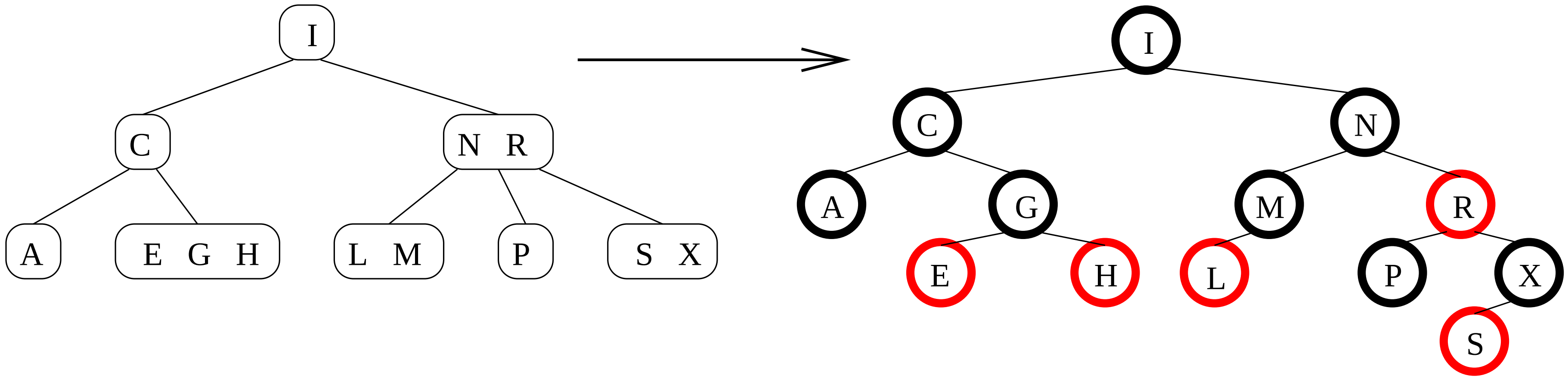
- The extra bit identifies a node as red or black
- A red node always has a black parent (the root is never red)
- Red nodes bind with their black parent to form a 3-node or 4-node



Red-black trees: example

Every valid red-black tree has an equivalent 2-3-4 representation

- Thus we have an extra restriction on valid red-black trees:
  - Must be an equal number of black nodes on every path from the root to a node with fewer than two children



- Note that difference between longest and shortest leaf-node path lengths is never greater than a factor of two
  - Equal number of black nodes, each of which may have a red child

Red-black trees: benefits

The major benefit of red-black trees is that the search algorithm is identical to that of a standard BST

- It ignores the extra colour information
- It does not modify the tree (unlike a splay search)
- So it will be *very fast*

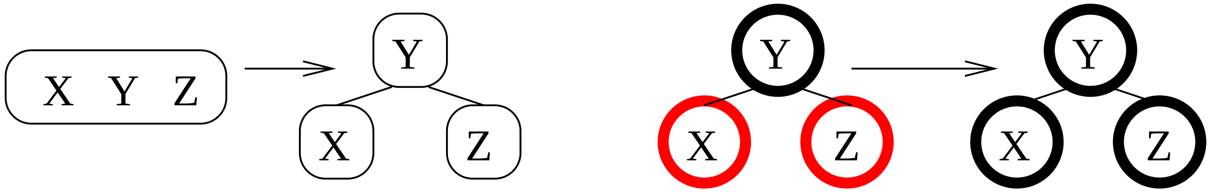
Furthermore, every basic operation executes in worst-case $O(\log N)$ time

- Because the longest path length is always less than $2 \log_2 N$
- All operations have a small constant of proportionality if implemented carefully

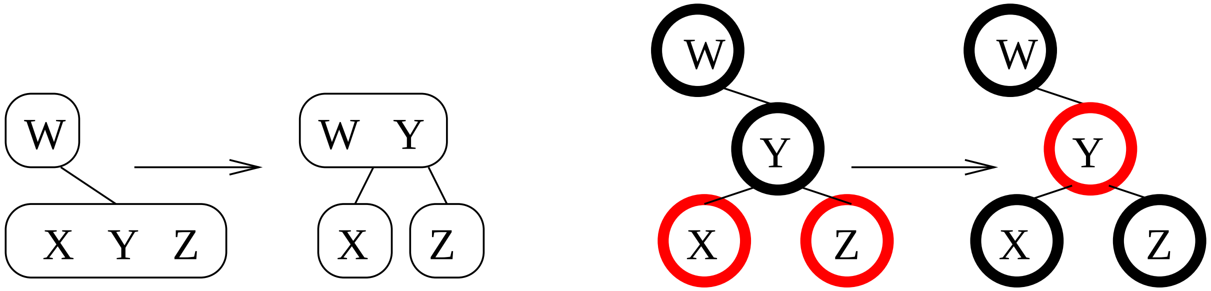
Red-black trees: insertion (1)

Insertion in a red-black tree can be implemented as a direct translation of the top-down 2-3-4 splitting algorithm

- To split a root '4-node': recolour the two children black



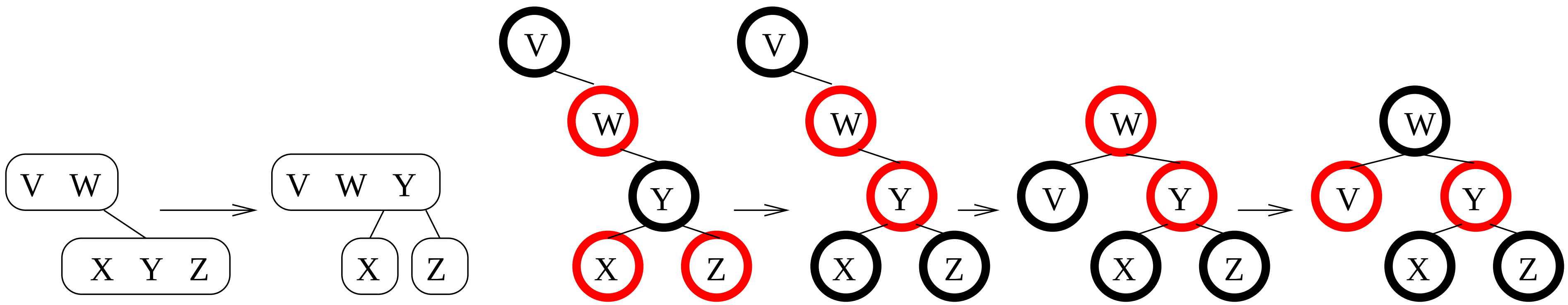
- To split a non-root '4-node': flip the colour of all three nodes



Red-black trees: insertion (2)

One problem is that we may end up with two adjacent red nodes

- If the parent is a ‘3-node’ oriented the wrong way round
- We fix this with a single rotation and further recolouring



We use this same transformation when we finally insert the new leaf node, if its parent is red (all newly-inserted nodes are initially coloured red)

- If the parent is black then no further transformation need be applied

Red-black trees: practical concerns

The direct translation of the top-down 2-3-4 insertion and deletion algorithms is not how you would implement red-black trees in practice

- We end up doing less splitting or merging if we optimistically leave the necessary transformations until *after* the insertion or deletion
- In the worst case we may end up making two passes over the tree (root-to-leaf search then leaf-to-root splitting/merging)
  - But this worst case is very rare
- Typically perform no more than a couple of splits/merges per operation
- These *bottom-up* implementations require recursion or (more usually) a parent pointer in each node

Implemented in this way, red-black trees have *excellent* performance: they are often the search structure of practical choice when a guaranteed performance bound is required.

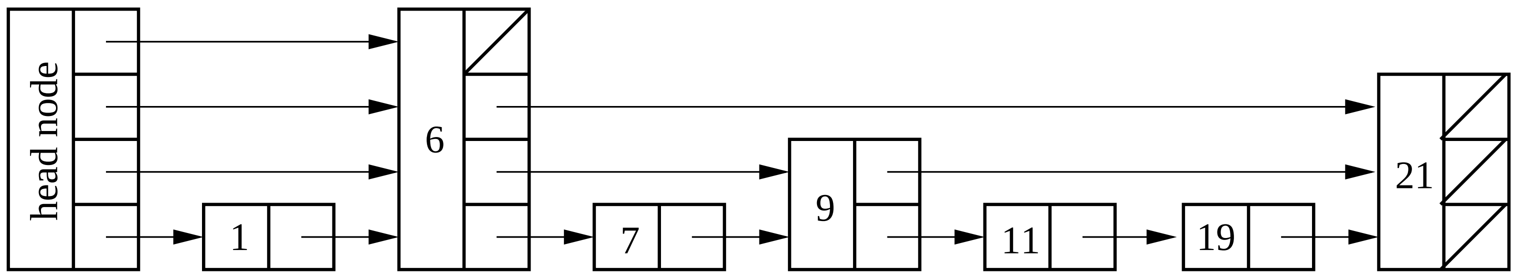
Skip lists

Skip lists are an alternative to balanced trees which, like BSTs, give a good average bound on running time

- But as we will see, the cost analysis is unaffected by the input

The idea behind skip lists is to augment the singly-linked list data structure with extra pointers to avoid the linear search time

- The pointers allow us to *skip* most items with a lower key value without needing to visit them individually



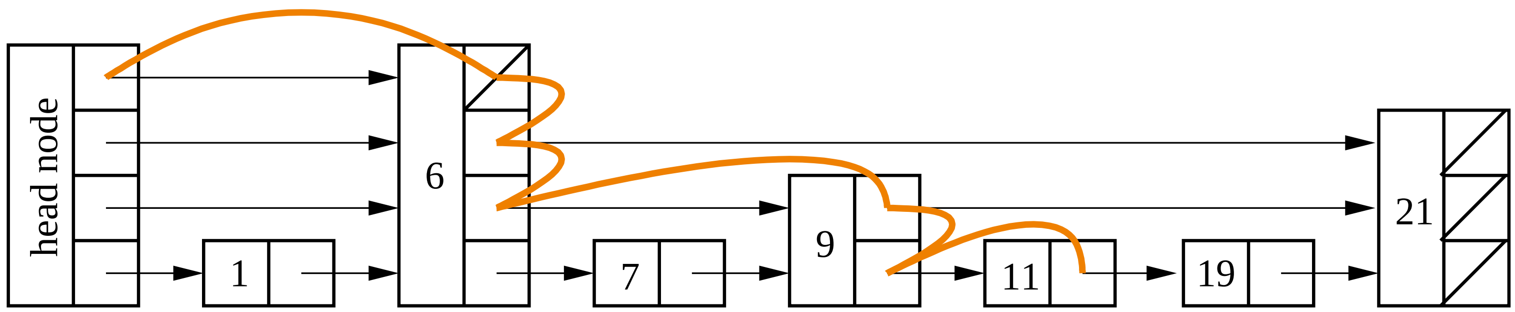
Skip lists: search

A skip list has k list levels, numbered 0 to $k - 1$

- The 0th level contains all nodes (it is an ordinary linked list)
- The ith level contains a subset of the nodes in level $i - 1$

Searches start the highest level and proceed until they reach or overshoot the search key

- The search then continues at the next lowest level
- ...and so on until it matches or overshoots at level 0



Skip lists: node representation

```
class Node {
    int height;
    Node[] next; /* height references */
    Item item;
}
```

Each node has a *height*, determined when it is first allocated

- A node of height h is inserted at all levels $< h$ in the skip list
- A height-h node has h forward pointers

The head node is a sentinel with a key value less than any that will be inserted in the list (effectively $-\infty$)

- It is the first node in every level of the skip list
- Thus it has the maximum possible height of any node

Skip lists: search implementation

```
Item search(Node head, int key) {
    Node curr, prev = head;
    /* Traverse each linked list in turn: top to bottom. */
    for (int i = head.height-1; i >= 0; i--) {
        /* Traverse the i'th linked list. */
        curr = prev.next[i];
        while ((curr != null) && (curr.item.key < key)) {
            prev = curr;
            curr = curr.next[i];
        }
        /* Did we find a match or did we overshoot? */
        if ((curr != null) && (curr.item.key == k))
            return curr.item; /* match: success */
    }
    return null; /* no match: fail */
}
```

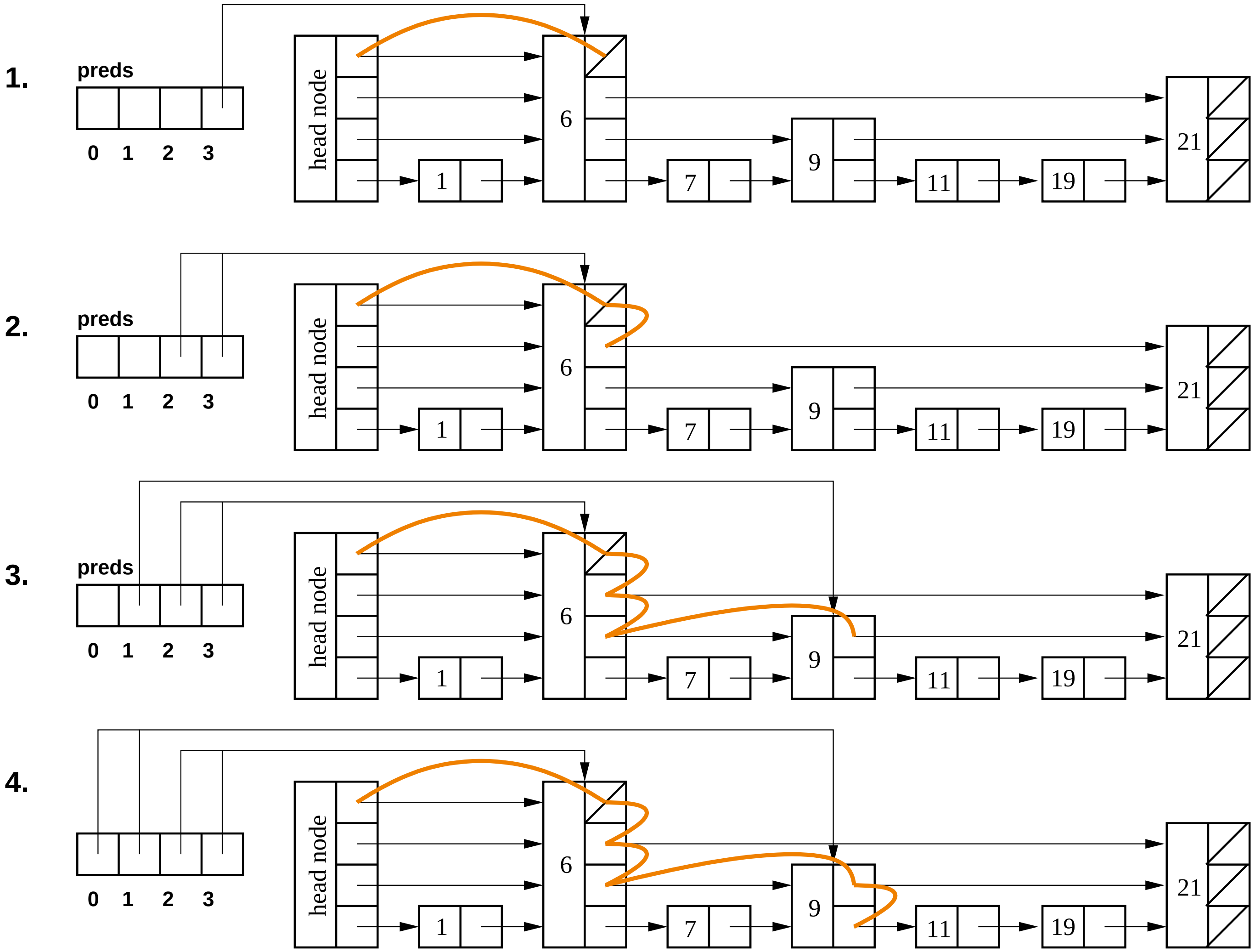
Skip lists: finding predecessor nodes

When inserting or deleting a node, we have to update each linked list in which that node exists.

- A helper function makes this easer: return the largest node at each level with key value $< k$.

```
Node[] find_preds(Node head, int key) {
    Node[] preds = new Node[head.height];    /* allocate result array */
    Node   curr, prev = head;
    for (int i = head.height-1; i >= 0; i--) {
        curr = prev.next[i];
        while ((curr != null) && (curr.item.key < key)) {
            prev = curr; curr = curr.next[i]; }
        preds[i] = prev;                     /* fill in result array */
    }
    return preds;
}
```

Skip lists: example of finding predecessors



Skip lists: delete

Deletion simply requires us to find the set of predecessor nodes

- We can then check for a key match by looking at the node x that follows the 'predecessor' at level 0
- If we have a match, then delete x from every level at which it exists

```
void delete(Node head, int key) {
    Node[] x, preds = find_preds(head, key);
    x = preds[0].next[0]; /* potential match */
    if ((x != null) && (x.item.key == key)) {
        /* Delete node at every level up to its height. */
        for (int i = x.height-1; x >= 0; x--)
            preds[i].next[i] = x.next[i]
    }
}
```

Skip lists: insert

Insertion is also straightforward: find predecessors, check for existence at level 0, and insert a new node if no match.

```
void insert(Node head, Item new_item) {
    Node[] x, preds = find_preds(head, key);
    x = preds[0].next[0]; /* potential match */
    if ((x != null) || (x.item.key == new_item.key)) {
        /* Existing match: update item reference. */
        x.item = new_item;
    } else {
        Node new_node = new Node(new_item);
        /* Insert new node at every level up to its height. */
        for (int i = 0; i < new_node.height; i++) {
            new_node.next[i] = preds[i].next[i];
            preds[i].next[i] = new_node;
        }
    }
}
```

Skip lists: performance

The performance of skip lists depends on how many nodes we can skip over on each comparison.

If level i contains half the nodes that are in level $i-1$ then each level contains approx $N/2^i$ nodes.

- **If** the subset of nodes present at each level is chosen randomly; and
- **If** the skip list has just enough levels that the top level contains $\Theta(1)$ nodes (ie. number of levels is $\Theta(\log N)$).
- ...then a skip list has average search time $\Theta(\log N)$

The analysis is non-trivial, but essentially we expect to visit $\Theta(1)$ nodes at each level, yet skip $\Theta(N)$ nodes for each visited node (at the higher levels of the skip list)

- The analysis depends on us randomly picking the height h of each node in the list. That is node height h is a r.v. with $Pr(h = k) = 1/2^k$

Skip lists: implementation tricks (1)

Dynamically choosing list height

Although it may seem we need to know the largest possible magnitude of N in advance, in fact we can grow the list height dynamically with N.

- eg. start with 5 levels, then add a level whenever $N = 2^{h+2}$
- Adding a level is as simple as incrementing the head node's height

Efficiently choosing a node's height

Implementing a random variable h such that $Pr(h = k) = 2^{-k}$ is actually very easy to do if you have a source of uniform random numbers

- Each bit in the binary representation of each random number is an IID r.v. with $50/50$ probability of being 0 or 1
- The probability of the first k bits all being 0 is 2^{-k} (one possible value in 2^k equally-likely values)

Skip lists: implementation tricks (2)

The probability of the kth bit (counting from 1) being the first non-zero bit is:

$$\begin{array}{rlcl} & Pr(\text{first } k-1 \text{ bits are } 0) & \cdot & Pr(k\text{th bit is } 1) \\ = & 2^{-(k-1)} & \cdot & 2^{-1} \\ = & 2^{-k} & & \end{array}$$

- So we can implement h as the index of the first non-zero bit in a number drawn from a uniform random number generator

The first non-zero bit can be found in a number of ways

1. Repeatedly shift the random number right by one bit position (ie. halve the number) and look at the least significant bit of the result.

2. Many CPUs provide a 'find-first-set-bit' instruction which executes in as little as a single processor cycle!

Hash tables

The data structures we've looked at so far have made use of the relative ordering of key values to arrange for efficient (logarithmic time) search

- A very different approach is to use the key value, or some arithmetic function of it, as a direct index into a table of data records

For example, if we know that all key values are in some small range 0 to $k-1$, we could simply maintain a k-length array of data items and index into that with the key value to achieve $\Theta(1)$ searches!

- But clearly this doesn't scale to large key ranges
- The usual approach is to apply a *hash function* to the key to transform it into an index into a table (of, say, M entries)
  - Maps from the domain of key values (potentially huge) to the range of table indexes (much smaller)

Hash tables: simple example

Assume a table with ten entries and the hash function $h(k) = k \bmod 10$

		72		14		6			59

Since the range of the hash function is smaller than its domain, many key values will hash to each table entry

- This is called a *hash collision*
- Consider adding key 24 to the above hash table

Avoiding too many collisions is key to good hash-table peformance

- Pick a good hash function so that collisions happen rarely
- Allocate a large enough table so that it never becomes too full

Hash table algorithms are largely defined by how they handle collisions

Hash functions

A good hash function maps key values uniformly and randomly into the hash-table slots.

- Often you will know that the input keys are reasonably random, and a simple hashing function will suffice (eg. low-order bits of the key)
- Where this isn't the case, a function that returns random values regardless of patterns in the input distribution of keys should be chosen

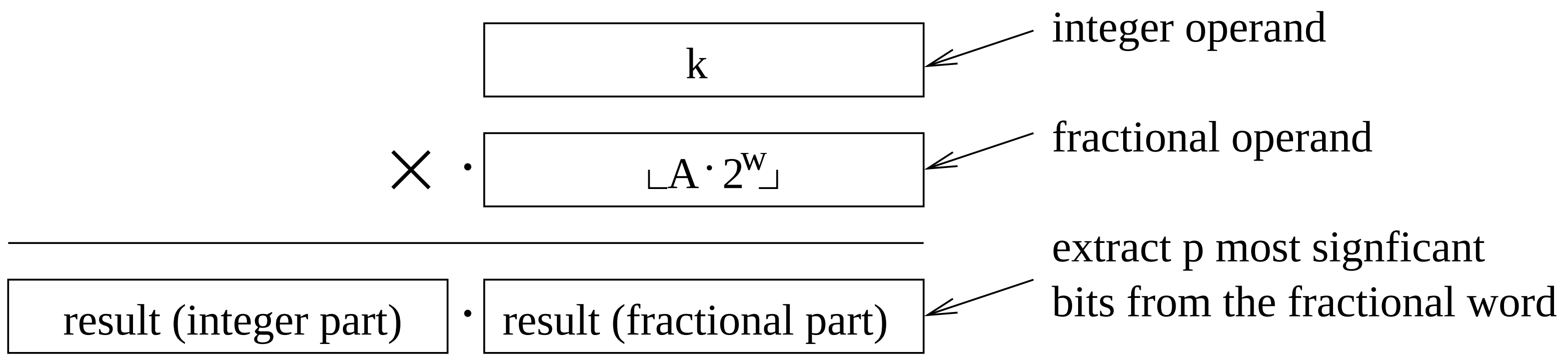
Division method: $h(k) = k \bmod M$
This works reasonably well if M is not a power of two: this would be equivalent to taking the low-order bits of k. We would like h to consider all bits in k. Often M is chosen to be prime.

Multiplication method $h(k) = \lfloor M(kA \bmod 1) \rfloor \quad (0 < A < 1)$
Multiply k by a fraction and extract fractional part. Hope this is uniformly distributed between 0 and 1.

Hash functions: multiplication method

The fractional arithmetic is fast if we used fixed point arithmetic and M is a power of two (say 2^p).

- Assume the machine word size is w (eg. $w = 32$ on a 32-bit system)
- We can precompute $\lfloor A \cdot 2^w \rfloor$. This is simply the w most significant bits.
- $k \lfloor A \cdot 2^w \rfloor$ is a simple multiplication of two integers.
- We can extract p bits from the result to quickly obtain $h(k)$.



Hash tables: open addressing

An *open-addressed* hash table stores all items inside a table of M buckets

- Typically each bucket holds up to one item
- So we must have $N < M$ (or dynamically grow M as required)

If the first access (*primary probe*) into the hash table does not find the required data item, then we perform a sequence of *secondary probes* on further buckets until we either find a match or find an empty bucket

- We can define our hash function as $h(k, i)$: parameterised both by key value k and by probe number i.
- The simplest probe sequence is *linear probing*: the ith probe is at $h(k, i) = (h'(k) + ci) \bmod M$. Frequently $c = 1$ and we simply probe every bucket in ascending order.

Hash tables: linear probing drawbacks

Although simple, linear probing leads to **primary clustering**

- Runs of occupied slots build up, increasing average search time
- If we have a half-full table with every other bucket occupied, average unsuccessful search probes is 1.5
- If the keys are clustered in one half of the table, average number of probes is $N/4$

h(k) = k mod 10

	22	33	2	13			7	38	27

Note that we get this effect even when $c \neq 1$ (ie. we increment by more than one on each probe)

- The primary clustering is simply a little better hidden

Hash tables: quadratic probing

Avoid making the slot address merely linearly dependent on the probe number: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod M$

- Generally much better than linear probing, but...
- ...need to pick c_1 and c_2 carefully to ensure that the probe sequence visits every slot in the hash table
  - At least c_1 and c_2 should be coprime to M
- ...and we still get **secondary clustering**: keys with the same initial hash value $h(k_1) = h(k_2)$ have the same probe sequence

Hash tables: double hashing

An even better method is to base the hash function on two sub-functions which define the probe-sequence start and probe-sequence step value:
$h(k,i) = h_1(k) + ih_2(k)$

In general we would like to choose h_1 and h_2 such that
$h_1(k_1) = h_1(k_2) \;\Rightarrow\; h_2(k_1) \neq h_2(k_2)$

- This will ensure that we avoid secondary clustering

Picking a pair of hash functions is quite easy if using the division method:

- $h_2(k) = 1 + (k \bmod M - 1)$ is a good choice: guaranteed to visit every slot if M is prime

In other cases we need to ensure that $h_2(k)$ is always coprime to M.

- Perhaps use a $h_2'(k)$ to index into a table of precalculated values

Hash tables: deletion

Deletion is surprisingly difficult in an open-addressed table

- If we simply make the slot empty, searches will terminate if they probe the slot even though their key may reside later in the probe sequence

h(k) = k mod 10

	22	~~33~~	2	13			7	38	27

Possible solutions:

- Find all keys that are in the truncated portion of a probe sequence and rehash them (may need to scan every bucket!)
- Place a *sentinel* value in the bucket that does not stop a search but indicates that an insertion may re-use that bucket
  - Sentinel values will accumulate, and worst-case search degenerates to $O(N)$. May need to periodically re-hash the whole table.

Hash tables: open-addressing performance

We define $\alpha = N/M$ as the *load factor* of the hash table

The expected number of probes for an unsuccessful search is $1/(1 - \alpha)$

- Expected number of probes increases rapidly as load factor approaches 1

Generally it is a good idea to keep α within a range 0.25 and 0.75

- Any less loaded has significant space overhead
- Any more loaded has significant time overhead

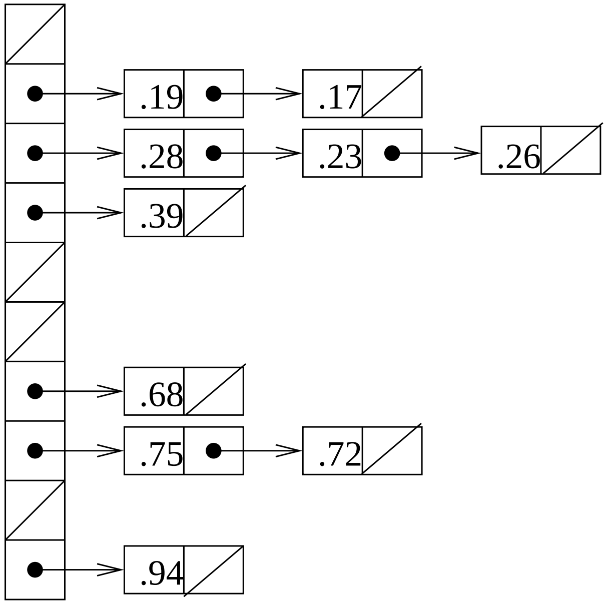
If you do not know N ahead of time, expect to dynamically grow the hash table periodically

- Allocate a larger table (eg. twice as large)
- Re-hash every item into the new table (expensive, but amortised across previous $O(N)$ insertions)

Hash tables: separate chaining

Each table entry (or *bucket*, to use the proper hash-table terminology) refers to a linked list of items that hash to it

Consider a hash table storing real-valued keys in the range $[0, 1)$ with hash function $h(k) = \lfloor 10k \rfloor$:



Note that this is the same data structure that we used to implement bucket sort, except here we do not (necessarily) care to maintain the collision lists in sorted order.

Hash tables: performance of separate chaining

The performance of separate chaining is good, even when there are more items in the list than there are unique buckets to locate them in.

So long as $N = O(M)$ and the hash function is well chosen, it is extremely unlikely that any one collision chain will get much larger than the rest.

This is usually a more popular approach than open addressing:

- No worry about the hash table becoming full.
- Performance degrades gracefully (linearly) as N increases.
- Performance is almost certain to be good as long as N is only a reasonable constant factor larger than M.
- Deletion is not difficult.

Hash tables: practical considerations

Hash tables are frequently the data structure of choice when:

- You are prepared to risk unlikely but rather bad worst-case performance
- You have some idea how big N is (since growing the table is a potentially very expensive, albeit occasional, operation)
- You do not care about the relative ordering of key values (efficient `select` and `sort` methods)

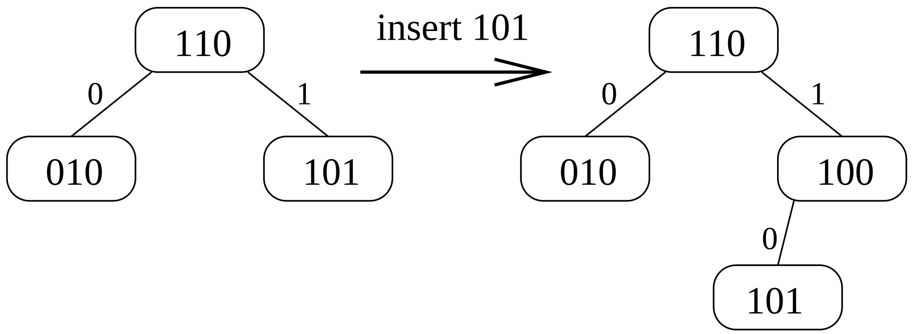
The third point is worth bearing in mind. Where you care about being able to select a range of adjacent data items, or scan all data items in order, or pick the kth smallest, an ordered data structure is the correct choice.

Radix search

Another approach to searching is to examine a section of the search key at a time

- The searching equivalent of radix sort

A *digital search tree* is a binary tree which branches on the value of successive bits of the search key.



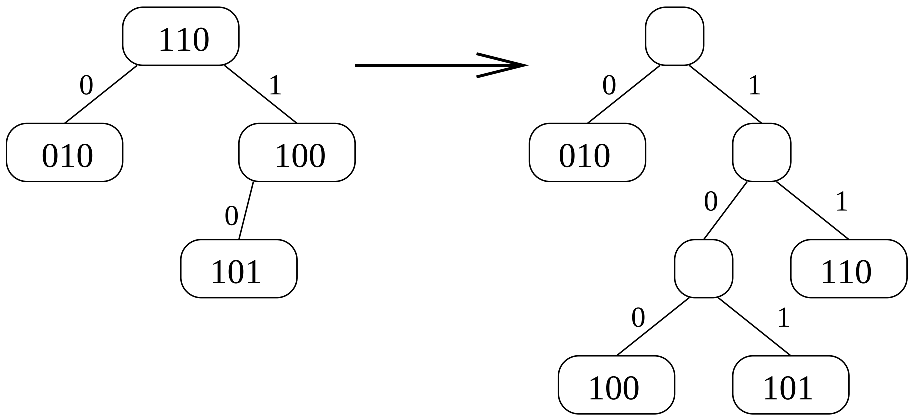
- The implementation of search/insert is identical to that of a BST except for the branching decision

Tries

Unfortunately the search tree this constructs does not make it easy to traverse data items in sorted order.

A *trie* solves this by storing data items only in leaf nodes.

- The path to a leaf node is the shortest key prefix which uniquely identifies the stored data item from all others in the trie.



- Not only can `sort()` be implemented by in-order traversal, but the representation for a set of key values is unique!

Trie performance

Tries are particularly effective when keys are long (eg. text strings)

- Avoid the cost of a key comparison at every level in the tree
- A trie only looks at the shortest necessary key prefix to find the unique possible match
  - A single full key comparison then confirms the match.

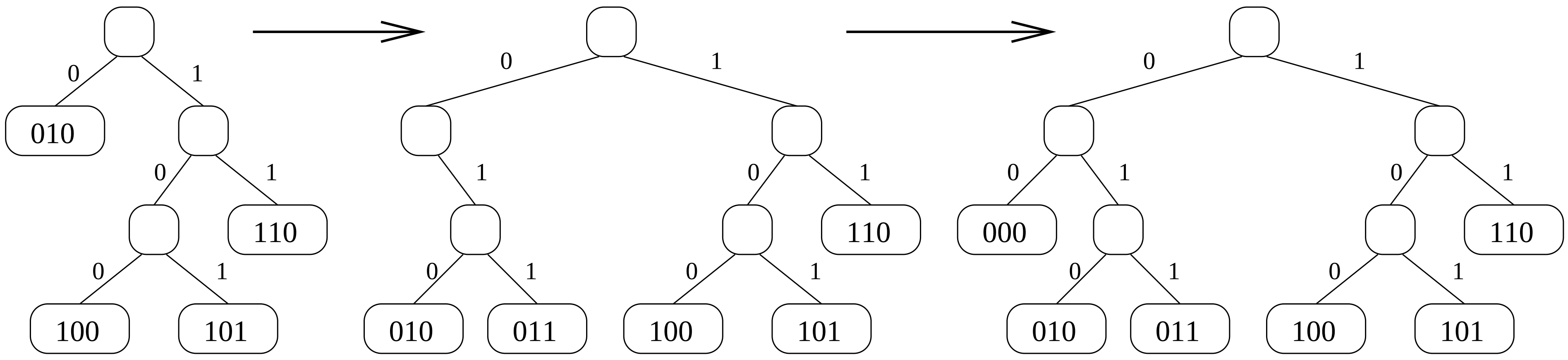
Tries can appear very unbalanced, if they contain lots of keys with similar prefixes, but the imbalance is bounded by the maximum key length.

Overall, the performance compared with a BST depends on key length, number of items being stored, and the distribution of key values.

Trie insertion

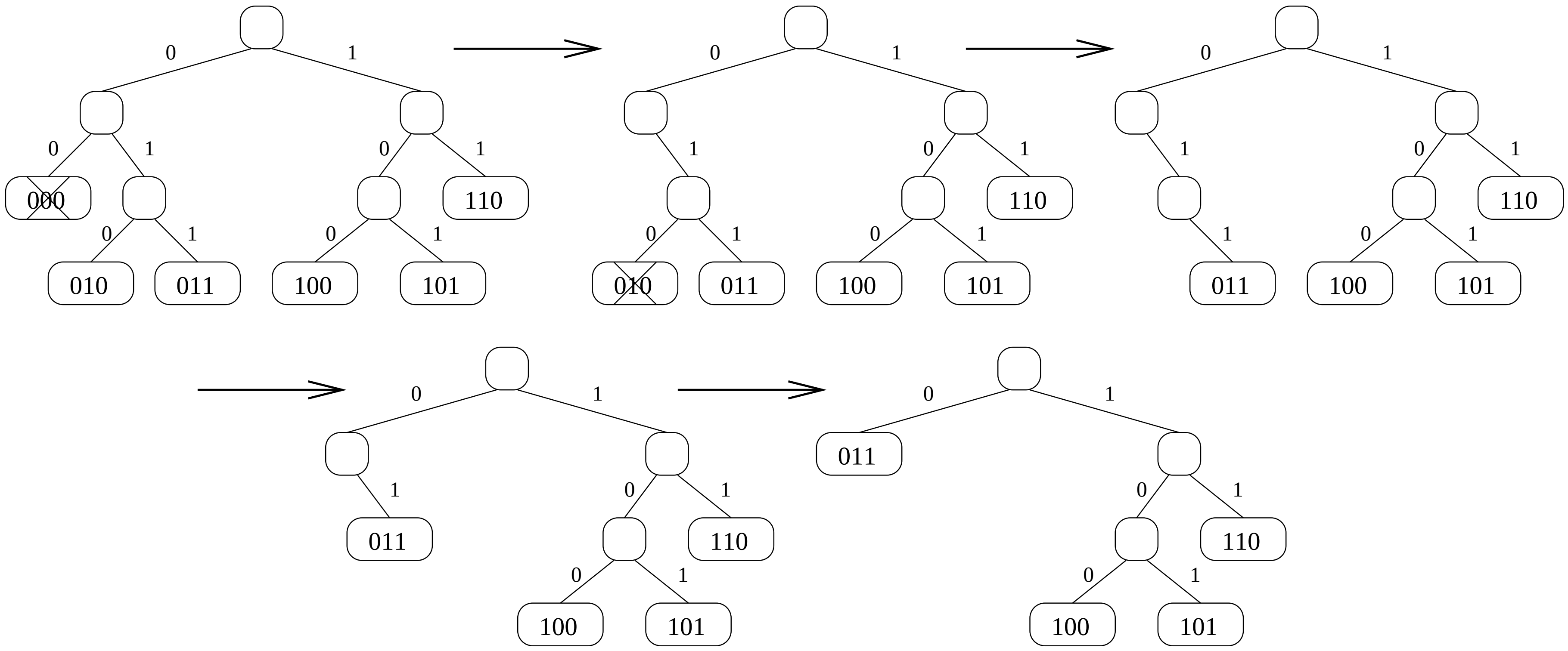
Search until reach an empty subtree of an internal node, or a leaf node.

- Empty subtree: simply insert a new singleton subtree
- Leaf node: insert enough new internal nodes to disambiguate keys



Trie deletion

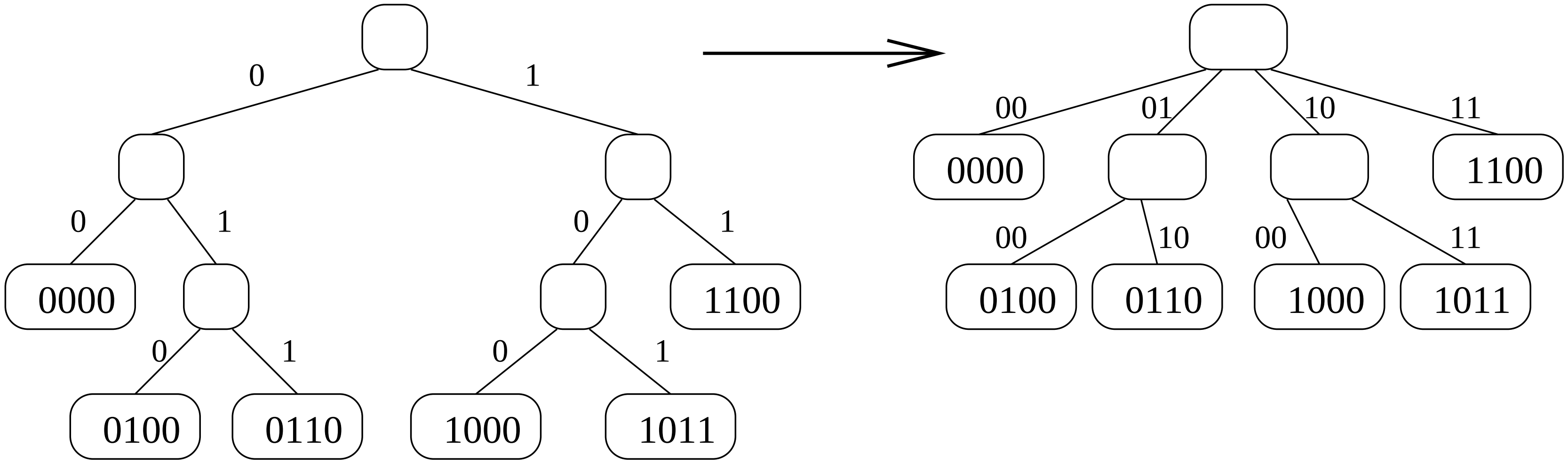
Delete the matching leaf node, then iteratively delete ancestor internal node(s) which have no subtrees or only one leaf node.



Multi-way tries

A common optimisation to reduce path lengths is to branch on more than a single bit at each internal node

- Use k bits to index into the node, rather than just one
- Each internal node contains an array of 2^k subtree references



Multi-way tries: performance

Indexing with k bits at each node reduces path lengths by a factor of k

- Unlike a comparison-based search tree (eg. 2-3-4 trees) increasing the branching factor does not increase the work per node for searches
  - Simply index into the array of subtrees: constant time
- Fewer nodes visited also improves cache locality
  - Poor locality is one of the drawbacks of binary trees (visit many little nodes rather than a few bigger ones each of which better fits in a cache line)

However, the larger internal nodes in a multi-way trie can be wasteful depending on the distribution of keys

- Would like to avoid too many empty subtrees

Finally...

Good luck

(And please fill in the on-line feedback form!)