COMP6700/2140 Trees

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

May 2017

Topics

Trees

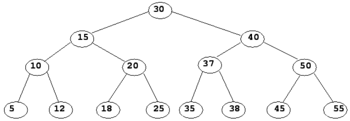
Binary Tree Data Structure

Trees are recursive data structure with more than one (two for *binary trees*) self-references. Of all standard DS they are most complex and interesting.

```
class Tree<E> {

    E stuff;
    Tree<E> left, right;

    public Tree(E e, Tree l, Tree r) {
        this.stuff = e;
        this.left = l;
        this.right = r;
    }
}
```



There are different types of binary trees (see next slide) which differ by how the elements are arranged and what operations are defined. The most commonly used is a balanced *binary search tree*. This structure is defined recursively (as it should), and it enables a binary decision (see below) to be made at each level which reduces the search time to an $\mathcal{O}(\log_2 N)$ operation (similar to the binary search in a linear ordered array).

Types of Binary Trees

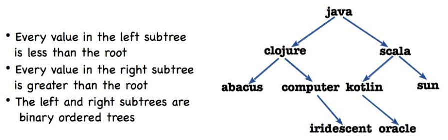
Multivalent trees are almost always can be reduced to binary ones (albeit with more complex node structure). Of all data structure considered so far, binary trees are the most complex, diverse and interesting (for research). Their value as DS is due to the fact, that the tree traversal is a key element of every operation, and for a well structured (balanced search, or amortised) tree, the traversal only consists in monotone descent, it takes $\mathcal{O}(h)$ steps (h is the height), and $h \sim \log_2 N$.

There are many types of tree DS (all listed below are binary except for B-Trees):

- Binary Search Trees (also known as Binary Ordered Trees)
- Heaps — (almost) complete BT which are built to represent an array and used for sorting
- Extensions of simple Binary Trees:
  - Red-Black Trees — binary trees where every node has one additional bit of data, ("red" or "black"), and the tree structure is subject to a colour-constraint; every operation (read, add, delete) must maintain this constraint (with an additional cost $\mathcal{O}(1)$). This ensures that the branches difference in length does not exceed the factor of 2, and the tree remains *approximately balanced*. (Java's API *TreeMap* implementation uses Red-Black Trees)
  - Splay Trees — an example of so-called self-amortising trees; after an access operation (usually, a read) it pulls the accessed node to the top, such that successive reads of the same value only take $\mathcal{O}(1)$. A sophisticated analysis allows to prove that the *average performance* over many reads is amortised in this way to the tree-ideal of $\mathcal{O}(\log_2 N)$.
- Expression Trees — where every node represents an operator, and every leaf is a value
- B-Trees — used in out-of-core algorithms (when data is so large, that it doesn't fit into memory, and had to be operated while on the disk)
- Many-many more-more

Binary Search Trees

Binary Search Trees (BST) are simplest case. They are used in implementation of *SortetSet* (*TreeSet* class) and *SortedMap* (*TreeMap*) interfaces.


• Every value in the left subtree is less than the root
• Every value in the right subtree is greater than the root
• The left and right subtrees are binary ordered trees
java
clojure
scala
abacus
computer
kotlin
sun
iridescent
oracle

BST are the most simple and often used type of binary trees. The example SearchTree.java defines the *SearchTree* class with the following operations:

- `get(E e)` — returns the node which contains e value (if found)
- `add(E e)` — adding an element while preserving tree's ordered structure
- `remove(E e)` — removes an element while preserving tree's ordered structure
- `height()` — returns the tree height
- `successor(E e)` — return the node with the value next larger to e
- `predecessor(E e)`— return the node with the value next smaller to e

Balanced Trees

It is important for a search tree to be *balanced* to exact the performance of its operations (listed on the previous slide) of $\Omega(\log N)$. Obviously, this requires the tree to be everywhere "dense", when every *branch* (path from the root to a leaf) has approximately the same value (maximum of which is called *height*). Such "dense" trees are usually called *balanced.* When balancing is broken some branches are abnormally long, with length $\sim N$. If a search tree becomes strongly unbalanced, the efficiency of its operations deteriorates to $\mathcal{O}(N)$.

Search trees are often build at run time, using the data which are streaming from an outside source (like it's done in the example SearchTree.java). If the source data is not skewed, the resulting search tree will likely be balanced, bit this cannot be guaranteed. If unbalancing occur, the tree can be *adjusted* to restore the balanced tree performance. This can be quite complex; examples of self-adjusting balanced trees include *Red-Black Trees* and *Splay Trees*.

There is a duality between algorithms and data structures. The *temporal structure of an algorithm* is dual to the *"spatial" structure of data*. When an algorithm performs in its generic fashion (with its "generic" efficiency, *eg*, *QuickSort* running at $\Omega(N \log N)$)), the structure of recursion calls (the temporal portrait, as it were) looks exactly as a balance search tree — everywhere dense without anomaly of protruding separate branches. This is how the "spatial" structure of a balanced tree look like, and it gives an optimal tree traversal performance $\mathcal{O}(\log_2 N)$.

Temporal structure of one resembles structural ("spacial") feature of another.

Heaps

There are two meaning of the term **heap** in Computer Science.

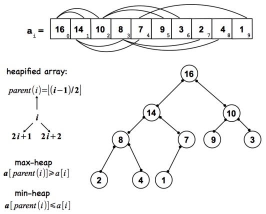
1. A part of computer memory which is unused by the OS and other running programs, and from which each program can borrow. In Java, all explicitly created objects use the heap memory. If you declare a fixed size array with elements like literal strings, integers, *enum* constants etc, the required memory is allocated in advanced. Otherwise, when a DS changes during the execution (when adding new elements), the memory comes from the heap. When an object is destroyed by the garbage collector, the memory is returned to the heap.
2. An array data structure `a[i], i=0..` in which elements are placed on an imaginary tree in accordance with the following index rules for an element of the index *i*:
   - Its parent's index is $\lfloor (i-1)/2 \rfloor$
   - Its left child index is $2i+1$
   - Its right child index is $2i+2$

   The heap element `a[i]` also satisfies a special condition — a so-called *heap property*:
   - $\mathtt{a}[\mathtt{parent(i)}] \geq \mathtt{a}[\mathtt{i}]$ — for *max-heaps*
   - $\mathtt{a}[\mathtt{parent(i)}] \leq \mathtt{a}[\mathtt{i}]$ — for *min-heaps*

   The array heap tree is always (almost) *complete* — the length of all branches is either equal to the tree height *h*, or *h-1* (this follows from the trivial mathematical fact that any number N satisfies the inequalities: $2^h \leq N \leq 2^{h+1} - 1$ for some positive integer h).

Heaps visualised

An array a of the length N can be "heapified" — the elements are given relationships between each other as prescribed by the heap property; the effect is like placing the elements on a tree. *What advantage does rearranging an array into a heap give?* Max-heaps are used in the sorting algorithm *HeapSort* which has $\Omega(N \log_2 N)$ *in-place* sorting performance. Its performance does not worsen like *QuickSort* on almost sorted inputs. Min-heaps are used in priority queues.



An array `a[i]` can be sorted by going through the steps:

1. Heapify a $\rightarrow$ `heap(a)`. The cost of this operation is $\mathcal{O}(N)$;
2. Sort the `heap(a)` using *HeapSort* (its efficiency $\mathcal{O}(N \log_2 N)$ is stable due to "heapness"). The factor N comes from the array traversal, and the factor $\log N$ is the cost of access ops, $\mathcal{O}(h)$, h is the height $h = \lfloor \log_2 N \rfloor$).

Self-Balancing and Amortising Trees

If writing (adding and deleting nodes) into a tree is allowed, its initially balanced structure can be compromised with the consequences of worsened behaviour — access cost $\mathcal{O}(h)$ (h is the tree height) instead being $\mathcal{O}(\log N)$ will be $\mathcal{O}(N)$. How to deal with this problem?

1. Add additional markers to nodes for constraining the structure; when the constraint gets broken (after `add` or `delete`), perform an local transform (called *rotation*) to restore the constraint. A well-known example is **Red-Black Trees**, in which every node (in addition to actual data) has a colour marker, “red” or “black”. The constraint:
   - the root is always black
   - all leaves are black, and all nodes (except the root) are either black or red
   - a red node children are always black
   - for each node, all monotone descending paths to leaves contain the same number of black nodes

   As mentioned above, the constraint guarantees that the tree remains approximately balanced — same node descending path lengths differ at most by factor of 2. Complexity of RBT structure and performance analysis is prohibitive for us here. A valuable study of Red-Black Trees by R. Sedgewick includes a Java code and animations.
2. As alternative to using additional markers, one can restructure the tree by moving a most recently accessed node towards the root. Such operation is called *splaying*, and the trees which support it are called **Splay Trees**. The case for using Splay Trees is when same elements are accessed *repeatedly* in close sequence (during which no large tree modification is performed). So despite an initial access can be costly, $\mathcal{O}(N)$, after splaying, it becomes $\Omega(1)$, and the average cost of one access gets *amortised*. It is possible to show that the average cost can be made the “usual” $\mathcal{O}(\log_2 N)$, but we shall not go into details.

Tries

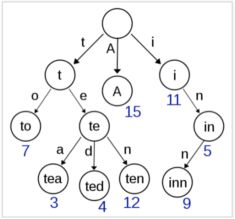
(pronounced as *tri:*, like “tree” only shorter *i:*)

The name *trie*s comes from reTRIEval in the retrieval (and storage) of information problem. They are also called *prefix trees*, or *digital trees*. It is an effective way to implement dictionaries (associative arrays) or sets. The data is organised in a tree form, but in a way different from a search tree. One particular implementation (by Phil Bagwell) — “Hash Array Mapped Trie” (*HAMT*) — plays an important role in one of the most promising JVM language *Clojure*.

A trie stores `(key,value)` pairs with keys “A”, “to”, “tea”, “ted”, “ten” “i”, “in”, “inn”, and values assigned arbitrarily (the root stores an empty string)

- values are associated with leaves and *some* nodes
- each key is defined by value’s *position* in the trie
- all descendants of a node have the same *prefix*

A trie can provide an alphabetical ordering of the entries by key and be used as an alternative to both hash tables and binary search trees:



- It has no collisions of different keys
- It does not have performance deterioration due to imbalance
- The worst look-up performance is $\mathcal{O}(m)$ (m is the length of a search string)

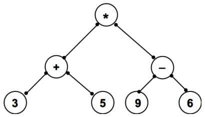
Expression Trees

The above examples dealt with tree structures to store data. Trees can also be used to represent expressions (including an entire computer program).

- an arithmetic expression, eg, (3 + 3) * (9 - 6) (the text is "compiled" into the tree below)
- a text, *eg*, a computer program, `Foo.java`

A tree which represents a computer program is called the *Abstract Syntax Tree* (AST).

Expression trees have operators for their nodes, and values for their leaves. A process which creates an expression tree from a text (an arithmetic formula like on the right, or a computer program) is called *parsing*. If the input string contains (syntax) errors, the parsing fails. Once built, an expression tree can be modified, optimised, transformed and evaluated.



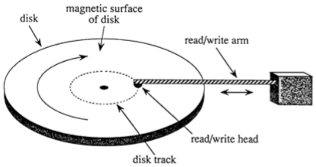
Compilation is an example of parsing process. Parsing is done via application of *grammar rules*, defined to substitute *tokens* of the input string onto syntactic categories (nodes of the parse tree) and terminal symbols (leaves). Grammar is a set of *recursive* relationships:

```
<expression> ::= <number> | <expression> <operator> <expression> | (<expression>)

<operator> ::= + | - | * | / | %
```

Out-of-Core Algorithms

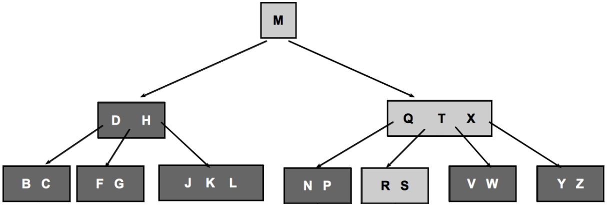
When data is too big to fit into memory and resides on a secondary storage device (hard disk, database *etc*) the time to access it is 50,000–100,000 times longer (from 50 nano sec to milli-seconds). If an algorithm involves a data structure from the "disk" — a so-called *out-of-core algorithm* — it must minimise read and write operations ("disk op") on such DS.



A very usable choice for on-the-disk DS is *B-Trees*. B-Trees are balanced like Red-Black Trees but the number of children at every node can change from a few to virtually thousands. The larger the number of children (let's say a B-Tree has t of them on average), the shorter the tree height: $h = \mathcal{O}(\log_t N)$, where N is the total number of nodes. Since the disk ops involve a node access, the fewer of them performed, the faster the algorithm runs.

The B-Tree Data Structure

The B-tree structure is defined as follows:



1. The tree node can have up to *(t-1)* sorted keys k_i which define t intervals: $[-\infty, k_1]$, $[k_{t-1}, +\infty]$ and $[k_{i-1}, k_i]$, $i = 1..(t-1)$.
2. When a value k is sought (inserted, deleted), the descent is performed into the subtree of a node which is represented by the i-th interval which contains $k \in [k_{i-1}, k_i]$
3. The maximum number of children t is determined by the size of memory (a *page*) which the RW head can "scoop" in one op. The scale of $t \sim 10^3$.
4. Standard `search`, `add` and `delete` operations can be defined with the performance $\Omega(\log_t N)$

Operations on Trees

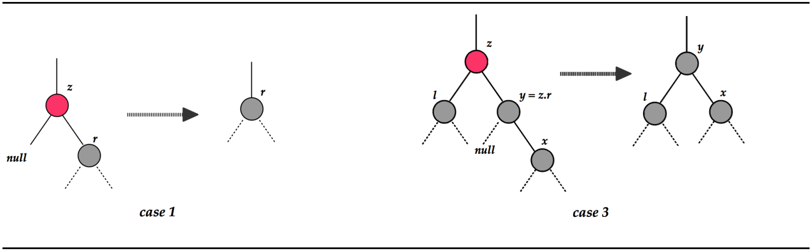
Just as for other kinds of DS, some operations are quite simple when they are defined recursively. For example, the search and return operation (call it `get(e)`) is called on the entire tree with the argument `e` of the type `E` data which is stored in tree nodes; if the node which contains the data equal to `e` is found, the tree node (which has the type `Tree<E>`) is returned, if not found — `null` value is returned. The operation `add(e)` must preserve the search-tree property: It should go down to one of the leaves and add a left or right subtree into which the new element `e` will be "deposited". The key aspect in both case is the tree *traversal*, and it is quite simple since we use the search-tree property:

- `get(e)` — start with the root node `this` and examine how its element compares with `e` by calling `this.stuff.compareTo(e)` (that's why we made *E* implement *Comparable*)
  - if `this.stuff.compareTo(e) == 0` — found it, return `this`
  - if `this.stuff.compareTo(e) < 0` — go left: `this.left.get(e)`
  - else — go right: `this.right.get(e)`
- `add(e)` — Start with the root (as alway) and, using comparisons to choose left or right turn at every node, traverse all the way down to a leaf where a new node containing the value `e` is attached. All new values are placed at one of the leaves at the bottom; this is the easiest way to preserve the search-tree properties while adding elements.
- `remove(e)` — Is hardest of all since the element `e` can be anywhere.
- `printAscend()`, `printDescend()`, `height()` — all recursive and simple whole-tree ops.

Deleting a Tree Node

The example SearchTree.java presents implementation of the first three operations in which recursion plays a vital role — `get(e)` and `add(e)` are downright recursive, and `remove(e)` relies on `get(e)` to the node `e`. Details of `remove(e)` are somewhat tricky.

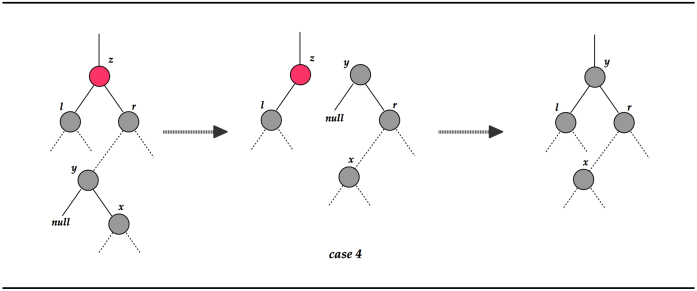
One has to consider four cases — three simple and a tricky one. When a node slated for deletion is found, it is pulled out of the tree, and its place is taken by another node which is carefully chosen in such a way that the resulting tree is *still a search tree*.



(Case 2 is not shown — it's similar to Case 1, just replace the right subtree onto the left one.)

Deleting a Tree Node (concl)

The most difficult case is generic one — when the deletable node *z* has a normal right subtree *r*, such that the right child is **not** the smallest element greater than that in the deletable node (like in Case 3). The delete operation requires a "surgery" to implant the node with smallest element *y* in place of *z* and rearranging broken links to preserve the search property. In more detail:



1. Find `y`, smallest in `z.r`
2. Extract `z` and move `y` in its place; `y.l` (formerly `null`) becomes `z.l`
3. former `y.r, x`, now takes place of `y` with all links carefully adjusted

It's worthy to study the code in SearchTree.java.

Non-Recursive Tree Search

Algorithms can be (non-)recursive, and DS can be (non-)recursive. Do all elements of this "2×2"-matrix make sense?

- Recursive operation on a recursive DS?
  - Define `get(e)` on a link list (an exercise for you)
  - Review BST operations which we've just reviewed (SearchTree.java)
- Recursive operation on a non-recursive DS (like an array)? Of course — recursive sorting algorithms like *QuickSort* and *MergeSort*, *BinarySearch*
- Non-recursive operation on a non-recursive DS? *SelectionSort*, *InsertionSort*
- Non-recursive operation on a recursive DS? More particular: a non-recursive traversal (search) on a binary tree?
  - Two important examples — *breadth-first search* (BFS) and depth-first search (DFS) are used for (complex) tree traversals when no facilitating property like search-tree is available, and one has to explore the entire tree. *Queue* and *Stack* local variables are used in BFS and DFS respectively as means of keeping the track inside the tree during the traversal (they are like Ariadne's thread in the Minotaur's labyrinth). Examples are in DepthFisrtSearch.java and BreadthFisrtSearch.java (these examples are not available this year).

Comparable and *Comparator*

Comparing objects (elements in a collection) can be done in two ways:

- class of stored objects implements *Comparable* interface (to support the `compareTo(Object o)`method)
- the constructor of *TreeSet* (or, *TreeMap*) is passed a Comparator<E> parameter (which has the method `int compare(E e1, E e2)`), if the elements do not implement *Comparable*, or if we want to compare them differently, and so we need to specify which comparison function to used for sorting the set.

```
class ItemComparator implements Comparator<Item> {
   public int compare(Item a, Item b) {
      String descrA = a.getDescription();
      String descrB = b.getDescription();
      return desrcA.compareTo(descrB);
   }
}
ItemComparator comp = new ItemComparator();
TreeSet<Item> sortByDescription = new TreeSet<Item>(comp);
```

In the example Library1.java, the same collection of *Book*s is first sorted by their natural order, and then by the the length of their title.

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient, Ch. 7.3, 7.4, 8.9, 8.10
- Oracle's Java Tutorial chapter on Map Interface and its Implementation