MPCS 51040 – C Programming

Lecture 7 – Hash Tables, Trees, Heaps

Dries Kimpe

May 8, 2017

Analysis of Algorithms

Definition

Study of the resource requirements (memory, CPU) of algorithms when applied to certain problems.

Why?

- Often multiple algorithms exist (sorting for example)
  - How to compare?
- Predict performance on different data
- Determine if there a faster algorithm is possible by comparing algorithm to theory

Worst-Case Analysis

The execution time of most algorithms depend on the exact data they are operating on. Example:

- Search: element might be at the beginning or end
- Sorting: data might be partially sorted

Most analysis of algorithms studies the *worst case* behaviour of the algorithm:

- Many algorithms often exhibit worst case (search)
- Unless you can predict how often the worst case behaviour will occur, known average or best case does not inspire confidence.
- For many algorithms tackling the same problem, best case is often the same.

O-notation

A mathematical description for expressing worst case behaviour; It limits the upper bound of a (mathematical) function within a certain constant factor:

Definition

if $g(n)$ is an upper bound for $f(n)$, this means that $\exists n_0, \exists c : \forall n \geq n_0 : cg(n) > f(n)$

In other words, for a large enough value of n and beyond, $g(n)$ will *always* be an upper bound, even though for smaller values of n, it might not be.

- The analysis focuses on the behaviour for *large* problems

If $g(n)$ is an upper bound for $f(n)$ we can say that $f(n)$ is $O(g(n))$.

O-notation

Some obversations

- Constant running time: $O(k) = O(1)$
  (this follows from the definition as you can pick your c of the definition to be k)
- Only the 'biggest' term matters, since, for large enough values, those will dominate.
  - $O(n^5 + n^4) = O(n^5)$
  - $O(kn^5) = O(n^5)$
  - $O(f1(n)) + O(f2(n)) = max(O(f1(n)), O(f2(n)))$

Examples

- Finding an element in an array is $O(n)$
  (with n the number of elements in the array)
- Sorting a list or array can never be $O(1)$. Why?

O-notation and Algorithms

Space vs Time

O-notation is typically applied for estimating the *time complexity* and *space (memory) complexity* of an algorithm.

How does O-notation relate to analyzing an algorithm (such as search)?

- It is hard to estimate how many 'steps' a statement or expression will take to evalute. Memory usage is often more explicit and easier to analyze.
  - but the exact number doesn't matter! (constants don't matter)
  - If a multiplication takes 100 cycles on one system, and 1000 cycles on another, the analysis is not affected. However, if the algorithm changes *the number* of additions as the size of the input data increases, than that does matter.
- Only loops *depending on the the data* matter; Otherwise the cost of the loop is constant (even though it may be large)!
- O-notation does not really give us information about the actual running time of the algorithm; Only about how the worst case scenario for that running time will change as the input data changes.

See tables 4-1 in the O'Reilly book for some common cases;
Insertion sort;

Set

Definition

set *Unordered*, *unique* (i.e. no duplicates) collection of members (distinguishable from each other).

Example

- $S_1 = 1, 2, 3$
- $S_2 = 1, 5, 2, 9$
- $S_2 \setminus S_1 = 5, 9$
- $S_1 \cup S_2 = 1, 2, 3, 5, 9$

Typical set operations:

- union
- intersection
- difference
- insert
- remove
- is_subset
- size
- is_equal

Sets

Implementation Notes

In O'Reilly Mastering algorithms with C, sets are implemented on top of linked lists

- A `match` function is provided when creating the set.
- The match only checks for equality. What it means for two values to be equal depends on the type... (Think about pointer to string vs string)
- Most operations are $\mathcal{O}(mn)$ with m and n the sizes of the sets involved.
  - Insert is $\mathcal{O}(n)$ (needs to check for duplicates)
  - Set difference, union, ... is $\mathcal{O}(mn)$

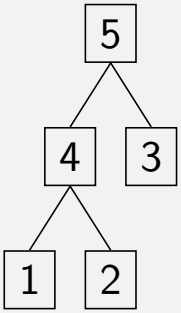
It easy to improve on this, while still building on top of a linked list.

- Change `match` to `compare`, enabling checking for *ordering* in addition to *equality*.
- Maintain the list in sorted order
  - Inserting is $\mathcal{O}(n)$ *worst case* (why?)
  - What is the complexity for difference, union?

Exercise: implement set difference using sorted lists.

Binary Tree

Tree Traversal



Definition

A binary tree is a tree where each node has at most two children. (The *branching factor* is 2)

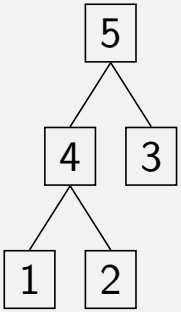
- 1 and 2 are *leaves* of the tree.
- 5 is the *root* of the tree.
- The *parent* of 2 is 4.
- 4 has two children: 1 and 2
- The *height* of the tree is 3

Full binary tree Binary tree where every node except for the leaf nodes has two children

Complete binary tree Binary tree where every level except for the last is complete, and all nodes on lowest level are on the left

Binary Tree

Tree Traversal



preorder node, then left, then right
5, 4, 1, 2, 3

inorder first left, then node, then right
1, 4, 2, 5, 3

postorder first left, then right, then node
1, 2, 4, 3, 5

levelorder Each level, starting at root, left to right
5, 4, 3, 1, 2

Binary Search Tree

Definition

A binary search tree (BST) is a binary tree for which the following property holds for each node:

- All nodes in the left subtree are smaller than the node.
- All values in the right subtree are larger than the node.

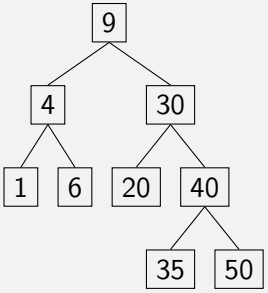
Note: Variations exist (for example allowing duplicate values)

Some observations:

- Where is the smallest element in the tree? The largest?

Binary Search Tree

Finding



How to search:

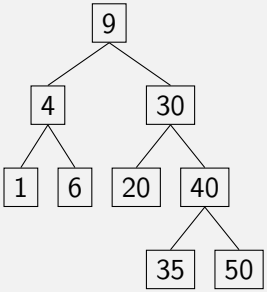
1. Start at the root node
2. Compare the current node against the element we're searching for
   - If equal: found, done.
   - If the node value is larger: continue with the left subtree
   - If the node value is smaller: continue with the right subtree
3. Repeat until found or running out of nodes

Search in a binary tree is: $\mathcal{O}(log(n))$ *average case* (i.e. assuming the tree is balanced).

Find element 20

Binary Search Tree

Insert



Example: insert 90

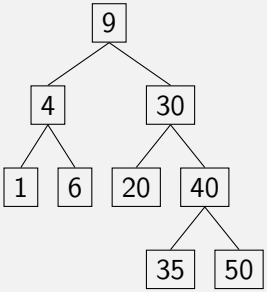
1. Do a find (i.e. where the element would be if it existed)
2. Insert where you would expect the element to be

Insert in a binary tree is: $\mathcal{O}(log(n))$ *average case* (i.e. assuming the tree is balanced).

Insert element 36

Binary Search Tree

Removal



Example: insert 90

1. Do a find (i.e. where the element would be if it existed)
2. Find the smallest element in the *rightmost subtree* of the node holding the element we want to remove
3. Swap the two elements
4. Remove the leaf element

Why does this work?

- Why does the swapping with the smallest element in the rightmost subtree not violate the BST property?
- Why will that element always be a leaf node?

Removal in a binary tree is: $\mathcal{O}(log(n))$ *average case* (i.e. assuming the tree is balanced).

Binary Search Tree

Implementation in C

```
1  typedef struct TreeNode
2  {
3     void * data;
4     struct TreeNode * left;
5     struct TreeNode * right;
6  } TreeNode;
7
8  // left or right are NULL if
9  // the subtree doesn't exist
10
11 // We need a comparison
12 typedef int (*compare_t)(void * d1,
13                           void * d2);
14
15 typedef struct BST
16 {
17    TreeNode * root;
18    compare_t compare;
19    // destroy func pointer etc.
20 };
```

- The nodes are linked together using pointers
- Same as linked list: data is typically `void *`
- Recursion is extremely useful with trees

Note:

- We need a way to compare two node values for equality and order

Definition

Heap: A complete tree (usually binary) which satisfies the heap property.

Definition

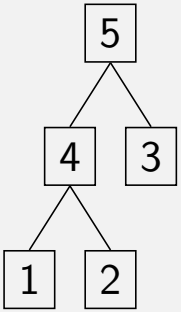
Heap Property: Each child node has a smaller value than its parent.

Note:

- Heaps are *partially* ordered; There is no defined order between the children or siblings.
- Heaps are complete trees: left-balanced, and all levels are full except for the last level.
- Heaps have minimum height of all trees with the same branching factor and number of elements.

Heap

Insert



How to insert in a heap:

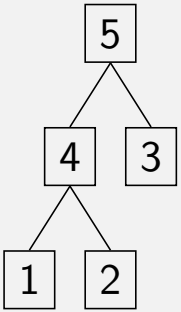
- Add an element at the next free position (as a leaf)
- Starting at the new element, ensure the heap property holds.
  - If smaller than the parent, swap child with parent (Note that the other child must be smaller than the existing or new parent. why?)
- Continue until reaching the root of the heap

Time complexity: $\mathcal{O}(log(n))$

Insert element in a heap

Heap

Remove



How to remove *the root of a heap*:

- Copy the last leaf into the root
- Starting at the new element, ensure the heap property holds:
  swap with the child which is the most out of order if needed.
- Continue until reaching an element which no longer violates the heap property

Time complexity: $\mathcal{O}(log(n))$

Note that we can *access* the root of the heap in constant time!
(What property does the root have?)

Extract root from heap

Array Representation

For a *zero indexed array*:

- The parent of a node at position i is located at $\lfloor \frac{i-1}{2} \rfloor$
- The children of a node at position i are at $2i+1$ and $2i+2$
- The root is at index 0
- The rightmost node at the lowest level in a tree with n elements is at index $n-1$

Array t: | 5 | 4 | 3 | 1 | 2 |

- Root is at index 0
- Children of the root: $0 \times 2 + 1$ (=1); $0 \times 2 + 2$ (=2)
- Parent of '2' (index 4) is at index 2
- Parent of '1' (index 3) is at index 1

Level order traversal!

Priority Queue

Definition

A priority queue is a set of elements in which the highest priority element (for varying definition of highest priority) can be quickly located and removed.

Priority Queue Operations:

insert	Insert element
peek	Return but not remove highest priority element
extract	Return and remove highest priority element
size	Return number of elements in queue

Priority queues can be implemented by on top of a sorted list; However, inserting can be $\mathcal{O}(n)$ worst case (when?). Better choice: use a heap:

peek	$\mathcal{O}(1)$
extract	$\mathcal{O}(log(n))$ worst case
insert	$\mathcal{O}(log(n))$ worst case

Inline functions

(Section 6.7.4 in C11 standard)

```
1  // inline linkage
2  inline int add(int x, int y)
3  { return x+y; }
4
5  // internal linkage
6  static
7  inline int add2(int x, int y)
8  { return x+y; }
9
10 void test()
11 {
12    // might call inline or external
13    // add function. Need to provide
14    // external linkage definition
15    // or can get linker error!
16    add ();
17 }
```

- `inline` is a *function specifier* which *hints* that the call to the function should be *as fast as possible*
- Rule 1: Any function with *internal* linkage can be an inline function.
- Rule 2: For a function with *external* linkage:
  - If declared with inline, it needs to be *defined* in the same translation unit.
  - If all file scope declarations for a function in a translation unit include `inline` without `extern`, than the definition is an *inline* definition which does not provide external definition or preclude an external definition in another translation unit.
  - Used to possibly provide an inline replacement for an external function (rarely used).
- Breaking these rules will likely cause linker errors (which might depend on optimization level).

`static` *in file scope* is to give the object (function or variable) *internal linkage*.
(i.e. symbol is only visible in the current translation unit)

Inline functions

The standard indicates it is a hint that calls to the function should be *as fast as possible*.

- *For example* (i.e. **not defined in standard**), by using *inline substitution*
  - Not textual substitution! (not a macro!)
  - Scope etc. not affected!
- Use case: small functions called repeatedly in loops
- Hint only: good compilers can decide to inline even non-inline marked functions.

There are drawbacks:

- Larger code size (which could cause slowdown)
- Slower compilation time (function is compiled repeatedly)
- Humans are very bad at determining when something will benefit from being inlined or not.

Whenever possible, inline functions should be preferred over macros

Smart usage of static (internal linkage) might be more beneficial!

Run-time vs Compile-time binding

```
1  int somefunc ();
2  int someother ();
3  ...
4  // Static (compile time) binding
5  somefunc();
6
7  int (*ptr)() = (someexpr ?
8          somefunc
9        : someother);
10
11 // Which function is called?
12 ptr();
```

- Regular functions are *bound* (i.e. decided) at *compile time*. The source code indicates which function to call, and the linker ensures that function gets called.
- Function pointers changed that
  - The decision which function to call is made at *run time*.

Polymorphism

```
1  // Java example
2  public interface Shape
3  {
4  public unsigned int area();
5  }
6
7  public class Square
8    implements Shape
9  {
10  public unsigned int area()
11  { return size*size; }
12 }
13
14 Shape a = new Square(2);
15 a.area(); // runtime binding
```

Polymorphism refers to the ability to process objects differently depending on their data type (or class).

- Example: library for shapes
- Every shape has an area
- Every shape has a name

Once a shape is created, the other code can use the shape without knowing exactly what shape it is.

Polymorphism in C

```
1  struct shape_t;
2  unsigned int shape_area(shape_t * shape);
3
4  shape_t * s = create_square(10);
5  shape_area(shape);
6
7  struct shape_t
8  {
9  unsigned int (*area)(shape_t * this);
10 void * data;
11 };
12
13 unsigned int shape_area(shape_t * shape)
14 { return shape->area(shape); }
15
16
17 // ---- square uses data as an unsigned int
18 static unsigned int square_area(shape_t * this)
19 {
20    intptr_t val = (intptr_t) this->data;
21    return val*val;
22 }
23
24 shape_t * create_square(unsigned int s)
25 {
26    shape_t * r = malloc(sizeof(shape_t));
27    r->data = (void*) s;
28    r->area = square_area;
29 }
```

- Run-time binding is once again obtained by using function pointers, grouped in a struct representing the interface.
- The *constructor* initializes the function pointers with the correct functions for this particular (derived) type (line 24).
- Helper functions (line 13) forward the operation to the corresponding function pointer.
- Type specific data is provided using a `void *` pointer which can be used by the derived types (line 10).
- *destructor* not shown.

Multiple ways of implementing polymorphism in C; This is just one example.

Recursion as a generator

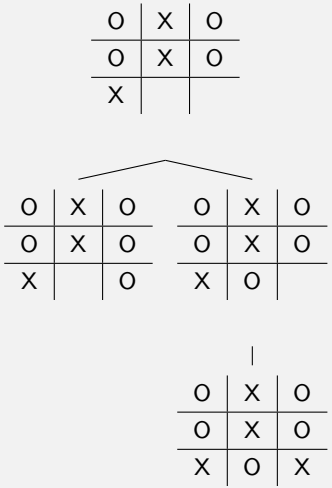
```
1  // print all OXO boards following from
2  // given board;
3  // board stored as character string
4  void generate(char * b,char place, int max)
5  {
6     // find empty spot
7     for (unsigned int i=0; i<max; ++i)
8     {
9        char c=b[i];
10       if (c!=' ')
11          // can't place anything here
12          continue;
13
14       b[i]=place;
15       // recurse
16       puts(b);
17       generate(b,
18                (place =='X' ? 'O' :'X'),
19                max);
20
21       // Remove token and try next pos
22       b[i]=' ';
23    }
24 }
25
26 char b[]="    ";
27 generate(b,'O',strlen(b));
```

- Typical recursion: define solution in terms of a reduced problem.
- Problem: generate all boards starting from given board.
  1. Do a valid move on each open position
  2. Generate all boards starting from the boards obtained after one additional move.
  3. Recursion end: when no more moves can be made.

We don't need to explicitly create or store the tree to enumerate all nodes

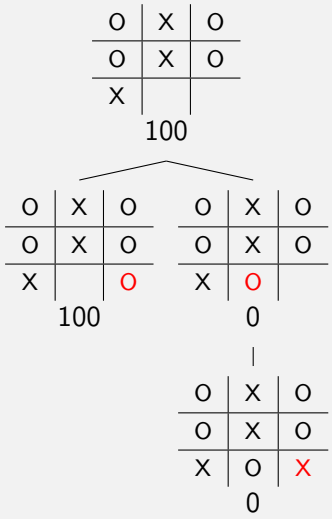
See `generate.c`

MiniMax Algorithm



Algorithm for two-player zero-sum games.

- Zero-sum: a good move for one player is an equally bad move for the other player. (Example: cutting cake)
- The algorithm tries to find the next move to make, by looking at all valid moves for the current situation and trying to estimate which move is the best.
- A move towards a victory condition is a good move, a move towards a losing game is a bad move.
- The algorithm (because zero-sum game) assumes that the other player will try to maximize their own score.

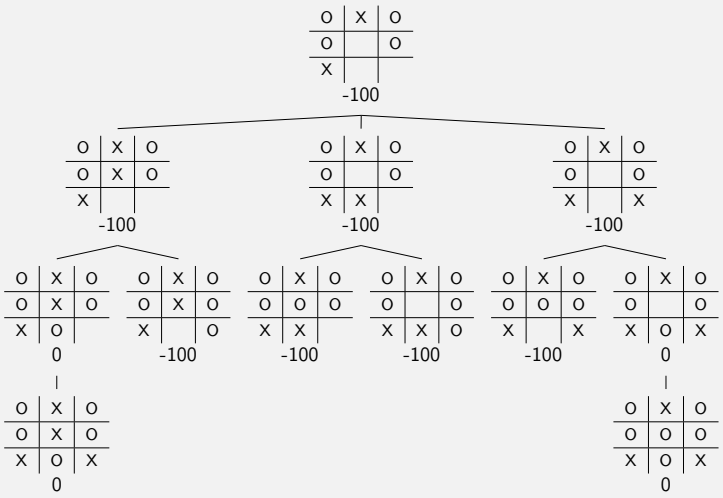


Algorithm:

1. Score the leaf nodes (end games); If the current player made the winning move, score it high. If the opponent made the winning move, score it low.
2. Score the intermediate nodes: if it is the current's player move, pick the best move (highest score) among the children. If the opponent is playing, assume it will maximize its own score and thus minimize our score: pick the lowest score among the child nodes.

Another example

Next move is X

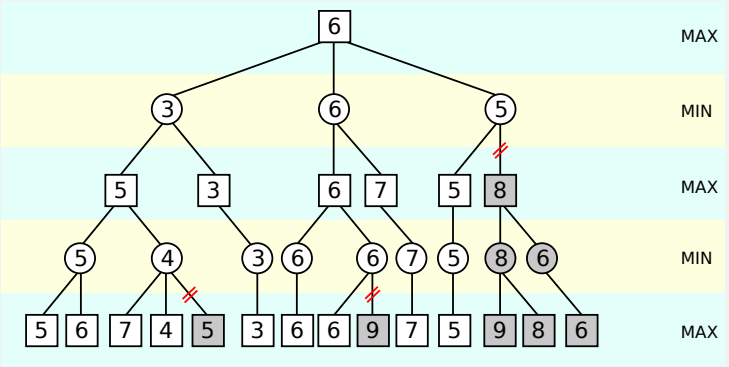


- First move is X
- MiniMax:
  1. Rate leaf nodes
     - 0 for tie
     - 100 for win
     - -100 for loss
  2. Rate children, picking best move if X plays, worst move if O plays.
- X lost this game, provided O plays optimal

MiniMax

alpha-beta pruning

Alpha-Beta pruning is an optimization to the MiniMax algorithm which will avoid evaluating a subtree if can be shown that the subtree will not be selected later on.



- Same principle in the win/loss/draw tree when a min in a min level is found, or a max in a max level.

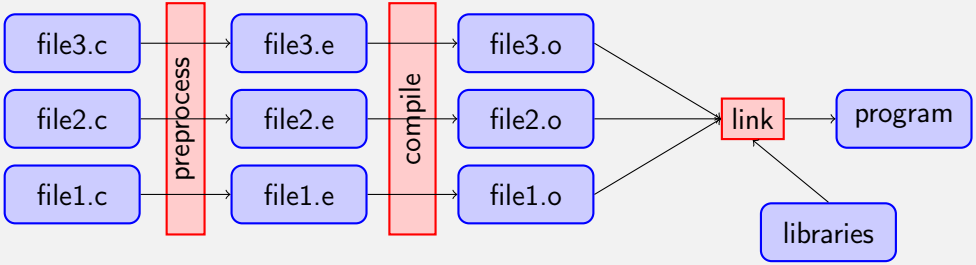
For information only – not needed for homework

Using Libraries

Reminder

Definition

Library Collection of *precompiled* code



Linking using gcc

Use the `-l` option on the final linking step

Example: `gcc -o myprog myprog.o -lcunit`

Assignment: Homework 6 & Reading

Homework Assignment

See `https://mit.cs.uchicago.edu/mpcs51040-spr-17/mpcs51040-spr-17/raw/master/homework/hw6/hw6.pdf`

Due next week – on Monday

HW6 info

Reading Assignment

O'Reilly Mastering Algorithms in C:

Required Chapter 8, 12