ESC190 "Stuff you should know" review

Brian Chen

Division of Engineering Science
University of Toronto
https://chenbrian.ca

`brianchen.chen@mail.utoronto.ca`

Contents

Section 1

Concepts you should be familiar with

By this point in the course you should be familiar with the c memory model as well as how to write programs using it. As a checklist, make sure you know the following well[1]

[1] *If not, we can start from this during the review session*

1. Pointers
2. `malloc`, `free`, `realloc`, `memcpy`, and so forth
3. Common memory errors and issues: invalid read, invalid write, segmentation fault, memory leaks, dangling pointers
4. Basic data structures and algorithms: linked lists, sorting, etc
5. Compiling c code, what a header file is, how to run code through a debugger, valgrind, include guards, and so forth
6. The idea of an abstract data type

Section 2

Abstract Data Types

Comment

All problems in computer science can be solved by another level of indirection
– Butler Lampson

Abstract Data Types (ADTs) are a way of organizing and structuring data in a program. They are abstract because they hide the implementation details of the data structure and only expose the operations that can be performed on the data.

Think of an ADT as a black box that you can interact with using a set of predefined operations. You don't need to know how the data is stored or manipulated inside the box, you only need to know what you can do with it.

In the following set of tutorial notes for hash tables, binary search trees, and AVL trees we will cover some concrete examples of ADTs.

Section 3

Heaps

Subsection 3.1

Definition

A binary heap is a binary tree that satisfies the *heap invariant*

Definition 1

Heap Invariant: The heap invariant is a property that must be satisfied by all nodes in a binary heap. In a max-heap, the heap invariant states that for every node n in the heap, the value of the node is greater than or equal to the values of its children. In other words, the maximum value in the heap is always located at the root of the tree.
Similarly, in a min-heap, the heap invariant states that for every node i in the heap, the value of the node is less than or equal to the values of its children. In other words, the minimum value in the heap is always located at the root of the tree.

It is also complete, meaning that all levels of the tree are completely filled except possibly for the last level, which is filled from left to right. We call a binary heap a *max-heap* if the total order is such that the root of the tree has the highest value, and a *min-heap* if the root has the lowest value.

Subsection 3.2

Operations

Considering a max-heap, a binary heap supports the following operations:

- `insert(x)`: Insert element `x` into the heap.
- `delete_max()`: Remove and return the element with the highest priority from the heap.
- `get_max()`: Return the element with the highest priority from the heap without removing it.

Subsection 3.3

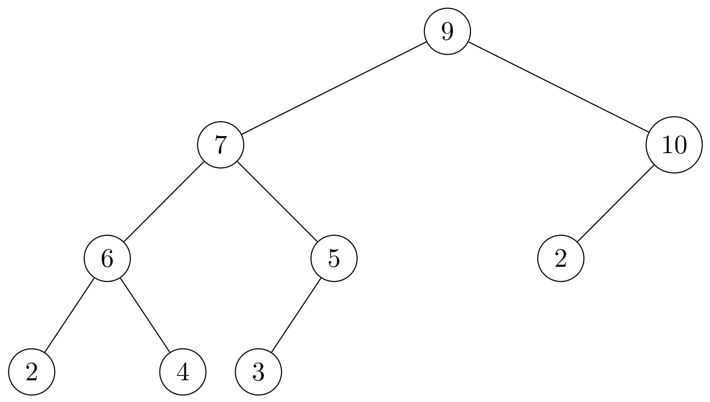
Implementation

Two common ways to implement a binary heap are using an array or using a binary tree.

However, since a binary heap is a complete binary tree, it is often easier to implement it using an array, where the children of the node at index i are located at indices $2i$ and $2i + 1$, and root is located at 1. This allows for efficient insertion and deletion of elements, as well as efficient access to the element with the highest priority.

1

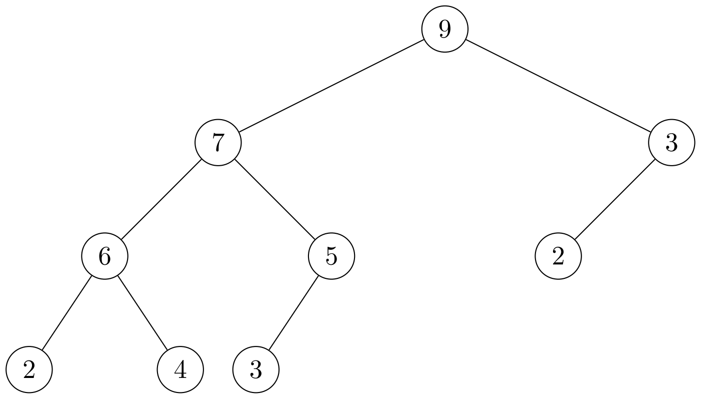
Is the following a valid binary max-heap?



Solution: In this example, the node with value 10 violates the heap invariant, as it is greater than its parent node with value 9. Therefore, this binary tree is not a valid binary max-heap.

2

Is the following a valid binary max-heap?



What is the array representation of this heap?

Solution: Yes, this is a valid binary max-heap because the heap invariant is satisfied for all nodes in the tree. The array representation is $[\emptyset, 9, 7, 3, 6, 5, 2, \emptyset, 4, 3, \emptyset]$.

Comment At this point it would be prudent to pause and make sure you are convinced that the heap invariant is satisfied for all nodes in the tree.

Subsection 3.4
Algorithms

For convenience we will define the following:

The following are for a min-heap. For a max-heap, the comparisons are reversed.

- Parent(i): The parent of node i. In an array, this is the element at index $\lfloor i/2 \rfloor$.
- Left(i): The left child of node i. In an array, this is the element at index $2i$.
- Right(i): The right child of node i. In an array, this is the element at index $2i+1$.
- Tail(H): The last element in the heap H. In an array, this is the element at index $H.size$.
- Root(H): The root of the heap H. In an array, this is the element at index 1.

Subsection 3.5
Insertion

To insert an element k into a heap H, we first add the element to the end of the heap, and then use a *heapify-up* operation to restore the heap property.

```
INSERT($heap, k$)
1  $heap.size = heap.size + 1$
2  TAIL($heap$) $= k$
3  HEAPIFY-UP($H, k$)
```

The HEAPIFY-UP operation compares the key of the newly inserted element with the key of its parent, and swaps the two elements if the parent's key is greater (for a max-heap) or less (for a min-heap). This process is repeated until the heap property is restored.

```
HEAPIFY-UP($heap, node$)
1  $parent = $ PARENT($node$)
2  while $parent \geq 1$ and $parent > node$
3      SWAP(ROOT($heap$), TAIL($heap$))
4      $node = parent$
5      $parent = $ PARENT($node$)
```

The HEAPIFY-DOWN operation compares the key of the root element with the key of its children, and swaps the two elements if the child's key is greater (for a max-heap) or less (for a min-heap).

```
EXTRACT_MIN($heap$)
1  if $heap.size == 0$ // heap is empty
2      return NULL
3  $heap.size = heap.size - 1$
4  SWAP($heap.root, heap.tail$) // swap the root with the last element
5  HEAPIFY-DOWN($heap, heap.root$) // restore the heap property
6  // return the minimum element (root), which is now the last element in the array
7  return $heap[heap.size + 1]$
```

Q: Why does the heapify-down operation not affect the min element after it is swapped? A: Because we decremented the size of the heap.

```
HEAPIFY-DOWN($heap, node$)
1  $child = $ MIN(LEFT($node$), RIGHT($node$)) // find the smaller child
2  while $child \leq heap.size$ and $child < node$
3  SWAP($node, child$)
4  $node = child$
5  $child = $ MIN(LEFT($node$), RIGHT($node$))
```

Comment

It's important to recognize that all of the analysis that we do is based on the assumption that the heap property is satisfied for all nodes in the tree. The insert operation disrupts the heap property, but the heapify-up operation restores it. Similarly, the extract-min operation disrupts the heap property, but the heapify-down operation restores it. By doing so, each operation maintains the *heap invariant*. You will learn more about this in future data structures and algorithms courses.

SUBSECTION 3.6

Complexity

The time complexity of the `insert` operation is $O(\log n)$, where n is the number of elements in the heap. The time complexity of the `extract_min` operation is also $O(\log n)$.

PROOF The height of a heap is at most $\log_2 n$, where n is the number of elements in the heap. The while-loops in the heapify-up and heapify-down operations are executed at most $\log_2 n$ times, because the height of the heap is at most $\log_2 n$. $\square$

It may be surprising that we can build a heap in $O(n)$ time[2]. This is because the order property in a heap is weaker than that of a sorted array or a binary search tree. In order to build a heap, we simply need to ensure that the heap property is satisfied for all nodes in the tree, not that the tree is ordered. A naive solution may be to call INSERT$(heap, k)$ n times. Alternatively we may just call HEAPIFY-DOWN$(heap, i)$ from the bottom up, starting at the last internal node[3]. This way, after each call to HEAPIFY-DOWN$(heap, i)$, the subtree rooted at i is a heap.

[2] *Since sorting an array is at best* $O(n \log n)$

[3] *Why not the leaf nodes? Because the leaf nodes are already heaps.*

```
BUILD_HEAP($array$)
1  $heap = array$
2  $heap.size = array.size$
3  for $i = \lfloor \frac{heap.size}{2} \rfloor \ldots 1$
4      HEAPIFY-DOWN($heap, i$)
```

The proof of time complexity of the `build_heap` operation is a little more involved.

PROOF HEAPIFY-DOWN$(heap, i)$ has a runtime of $O(h)$, where h is the height of the subtree rooted at i. A heap with n nodes has a height of at most $\log_2 n$ and for a level h there are at most $\lceil n/2^h \rceil$.

$$2^h \leq \frac{2^{\lfloor \log_2 n \rfloor}}{2^h} \leq \frac{n}{2^h} \tag{3.1}$$

So the cost of heapifying all subtrees is

$$T(n) = \sum_{h=0}^{\log_2(n)} O(h) \cdot \frac{n}{2^h} = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(2n) = O(n) \tag{3.2}$$

By taking advantage of the fact that

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{0}{1} + \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \ldots = 2 \tag{3.3}$$

This can be proven by using the geometric series and taking a derivative.

$$\begin{aligned} S &= \sum_{h=0}^{\infty} \frac{h}{2^h} \\ &= \frac{1}{2} \sum \frac{h}{2^{h-1}} \\ &= \frac{1}{2} \sum \frac{h-1}{2^{h-1}} + \sum \frac{1}{2^{h-1}} \\ &= \frac{1}{2}(S - \frac{1}{2^{-1}} + 4) = \frac{1}{2}(s+2) \Rightarrow S = \frac{1}{2}S + 1 \\ &= 2 \end{aligned} \tag{3.4}$$

$\square$

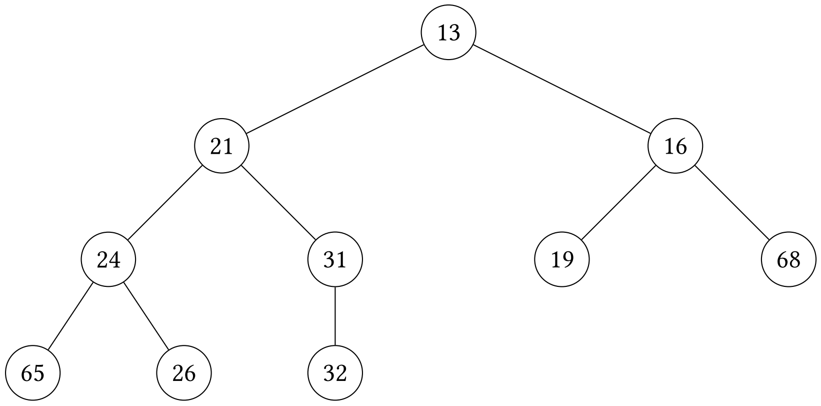
SUBSECTION 3.7

Example Problems

Comment I can either go through these during tutorial or you can do them on your own.

3

- Run the `build_heap` operation on the following array: $[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$. Draw the heap after each step.
- Run insert and extract_min on the following heap: $[\emptyset, 13, 21, 16, 24, 31, 19, 68, 65, 26, 32]$. Draw the heap after each step.



- How could we implement a priority queue using a sorted array? What would be the time complexities of the various operations? Are there situations where this might be favourable?
- Implement a priority queue using a binary heap in c. Do this with an array or a tree. What would be the time complexities of the various operations? Are there situations where one implementation might be favoured over the other? What would the `struct` declarations look like?

If we were to implement a binary heap using a binary tree, how would we store the elements in the tree? What would the struct look like? Would this have any impact on your algorithms?

Section 4

Dynamic Programming

Dynamic programming is a technique for solving problems by breaking them down into smaller subproblems and solving each subproblem only once. It is particularly useful for problems that have overlapping subproblems, meaning that the same subproblem is solved multiple times.

Dynamic programming is best learned by carefully studying examples until things start to click. Here are the steps involved in solving a problem by dynamic programming:

- Identify the subproblems: Break the problem down into smaller subproblems that can be solved independently.
- Define the recurrence relation: Define a recurrence relation that expresses the solution to each subproblem in terms of the solutions to smaller subproblems.

- Compute the solution: Use the recurrence relation to compute the solution to each subproblem in a bottom-up manner.
- Construct the solution: Construct the solution to the original problem from the solutions to the subproblems.

Although dynamic programming algorithms are easy to design once you understand the technique, getting the details right requires carefully thinking and thorough testing.

Another common dynamic programming technique is top-down memoization. Top-down memoization is a technique for implementing dynamic programming algorithms that involves recursively solving subproblems and storing the results in a memo table. The memo table is used to avoid recomputing the same subproblems multiple times. The memo table is typically implemented as an array or a hash table. The keys of the memo table are the inputs to the subproblems, and the values of the memo table are the solutions to the subproblems. Though easier to implement, top-down memoization tends to be less efficient than bottom-up dynamic programming, especially in terms of space complexity.

The difference will become more clear with an example.

Subsection 4.1
Fibonacci

4.1.1 Recursive

```python
def fibonacci(n):
    if n <= 2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

4.1.2 Top-Down

```python
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 2:
        return 1
    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
    return memo[n]
```

4.1.3 Bottom-up

```python
def fibonacci(n):
    if n <= 2:
        return 1
    fib = [0] * (n+1)
    fib[1] = 1
    fib[2] = 1
    for i in range(3, n+1):
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
```

Comment The bottom-up implementation uses an array to store the Fibonacci numbers, which is more space-efficient than the top-down implementation that uses a memo dictionary. That being said, we could very well use an memo array instead. For this problem it is not a significant difference but for problems with larger inputs or initial states it can make a big difference.

Subsection 4.2
Coin Change

See Qilin's wonderful notes: http://xueqilin.me/esc190-notes/part3.html#L28

Subsection 4.3
Cutting Fabric

Example For a theater play, L meters of fabric have to be purchased from at most n stores in integer quantities. Each store has a price p_i per meter, but also a discounted price d_i for purchases of r_i meters or more. Each store can sell at most f_i meters of fabric. You are tasked with buying exactly K meters of fabric for the play at the minimum possible price.

- (a) Let $C[l][i]$ be the minimum cost of buying l meters of fabric from the stores $\{1, 2, ..., i\}$ Express $C[l][i]$ as a recurrence and give a brief explanation for its correctness.
- (b) Describe the implementation of your solution clearly (pseudocode is not mandatory). Try to make it as efficient as you can; marks will be deduced for inefficient solutions. Analyze its running time and space complexity (correctness and termination are not required).

$$C[l][i] = \begin{cases} 0 & l = 0 \\ \text{PURCHASE-FABRIC(l, i)} & i = 1 \\ min\left\{(C[l-k][i-1] + \text{PURCHASE-FABRIC}(l-k, i)) \quad \forall\, k \in [0, l]\right\} & \text{otherwise} \end{cases} \tag{4.1}$$

```
Purchase-Fabric($l, i$)
1  if $l > f_i$
2      return INF
3  if $l < r_i$
4      return $l * p_i$
5  if $l \geq r_i$
6      return $l * d_i$
```

This problem can be solved with bottom-up dynamic programming. The base cases are given by when there is only one store and length is 0: if $l = 0$ no fabric is purchased so the cost is 0, and when $i = 1$ the problem is trivial, i.e. is equal to PURCHASE-FABRIC(l, 1). For $l > 0, i > 1$ we apply dynamic programming by noting that this problem exhibits the optimum substructure property. Given matrix C describing the minimum cost for each $l > 0, i > 1$ pair, we can take the cost of $C[l][i]$ to be $min\{(C[l-k][i-1] + \text{PURCHASE-FABRIC}(l-k, i)) \quad \forall k \in [0, l]\}$. In other words, this says that the minimum cost at l, i is given by the lowest cost of purchasing k meters of fabric from $i-1$ stores ($C[l-k][i-1]$) and $l-k$ meters from the i^{th} store. We also know that this will terminate because the loops on lines 2, 4, 7, 7, 8, and 10 are bounded by finite n, L, and k. This recurrence is applied for all ($l \leq L, i \leq n$) pairs. The solution to the problem is therefore given by $C[L][n]$

```
Lowest-Cost-To-Purchase-Fabric($L, n$)
 1  C = new array of shape (L+1) x (n+1)
 2  for $i = 1$ to $n$
 3      $C[0][i] = 0$
 4  for $l = 0$ to $L$
 5      C[l][1] = Purchase-Fabric(l, 1)
 6
 7  for $i = 2$ to $n$
 8      for $l = 1$ to $L$
 9          C[l][i] <- INFINITY
10          for $k = 0$ to $l$
11              C[l][i] = min{C[l][i], C[k][i-1] + Purchase-Fabric(l-k, i) }
12  return C[L][n]
```

Space complexity

This algorithm allocates only a few loop variables and an array of shape $(n+1)(L+1)$, so the space complexity is $O(nL)$.

□

Time complexity

- Purchase-Fabric runs in $O(1)$
- The recurrence described in Line 11 runs in $O(L)$ since it checks $k = 0$ **to** l, and $l = L$ in the worst case
- The nested loops on lines 7 and 9 iterate through all (l, i), of which there are $n \cdot L$

Therefore the total runtime is $O(L) \cdot O(nL) = O(nL^2)$

Subsection 4.4

Longest Palindrome Subsequence

Example Given a string S, find the longest palindrome subsequence of S, where a palindrome is a string that reads the same forwards and backwards.
For example, in the string abacabad, the longest palindrome subsequence is abacaba.

The problem can be solved using dynamic programming.

Comment What if we were to solve this problem using a non-dynamic programming approach? What would be the running time of your algorithm?

Let S be the given string of length n

The base case is when $i = j$, since a single character is always a palindrome. For the general case, we have two possibilities: either the first and last characters of the substring are the same, or they are different. We can therefore create a 2D DP array L of size $n \times n$, where

$$L(i,j) = \begin{cases} true & \text{if the substring } S[i:j] \text{ is a palindrome} \\ false & \text{otherwise} \end{cases} \tag{4.2}$$

$$L(i,j) = L(i+1, j-1) \text{ and } S[i] == S[j] \tag{4.3}$$

With base cases corresponding to one and two letter palindromes

$$L(i,i) = true \tag{4.4}$$

$$L(i,i+1) = (S[i] == S[i+1]) \tag{4.5}$$

We may then build up the DP array by filling in the base cases first, and then applying the recurrence relation to the rest of the array in a bottom-up fashion, i.e. going to three-letter palindromes and so forth The solution to the problem is then given by the entry in L where $L[i][j]$ is true and $|j - i|$ is maximized.

4

What is the time complexity of this algorithm? Space complexity? Can we improve?

Section 5
Graphs

Comment I will leave this for the next office hours since we just started covering graphs in class.

Section 6
Recursion

Comment TODO

Subsection 6.1
MergeSort

Subsection 6.2
Traversing recursive data structures

Subsection 6.3

Making change, but recursive

Section 7

Binary Search Trees

Comment: It has been brought to my attention that we haven't covered binary search trees in class yet, so these notes are incomplete.

Let's say that we are interested in building an *Abstract Data Type*[4] that offers us the following operations:

[4] *ADT*

$$\text{Insert}(x) \to \text{Inserts } x \text{ into the data structure} \tag{7.1}$$

$$\text{Delete}(x) \to \text{Deletes } x \text{ from the data structure} \tag{7.2}$$

$$\text{Search}(x) \to \text{Returns true if } x \text{ is in the data structure, false otherwise} \tag{7.3}$$

Let's call this ADT[5] a *SearchBag*. [6]

[5] *Specially it is a container ADT*

[6] I am calling it a *SearchBag* because it is a bag of elements that can be searched through. Generally something like this would be called a search tree but we will also be discussing non-tree-based implementations

Our first implementation will be with an unordered array

It should be clear to us that we can implement Insert(x) in $O(1)$ time by simply adding the element to the end of the array, since we are not concerned with maintaining any order. It should similarly be clear that we can implement Search(x) in $O(n)$ time by simply iterating through the array and checking if x is in the array. Likewise, we can implement Delete(x) in $O(n)$ time by iterating through the array and deleting the first instance of x that we find.

Comment: Note that there could be scenarios where each implementation is faster than the other. For example, the unordered array implementation might be faster than the ordered array implementation if we are inserting a large number of elements into the array and we rarely need to make lookups. Or if we tend to search for and delete old entries. Or the additional memory and complexity overhead of building a tree might not be worth it if we are only storing a small number of elements. The choice of data structure and algorithm is always a trade-off between time, space, and complexity, and is often dependent on the specific use case. You will learn more about this as well as tools you can use to capture 'average-case' runtime (e.g. amortized analysis and probabilistic methods) in future algorithms courses: `ECE358: Foundations of Computing` is offered in third year and I think `CSC473: Advanced Algorithms` is a really cool course.

From ESC180 we learned a useful algorithm – binary search.

Comment: If you don't recall, binary search is an algorithm that allows us to search for an element in a sorted array in $O(\log n)$ time by repeatedly dividing the array in half and checking if the element we are looking for is in the left or right half and searching accordingly

Our next implementation will be with an ordered list.

The three ADT operations can be implemented as follows:

- Insert(x) in $O(\log n)$ time by iterating through the list and finding the correct position to insert x while maintaining the order of the list. This can be done using binary search to find the correct position in $O(\log n)$ time, and then inserting.

- Search(x) in $O(\log n)$ time by using binary search to find the position of x in the list.
- Delete(x) in $O(\log n)$ time by first using binary search to find the position of x in the list

Our final implementation will be using a binary search tree. A binary search tree is a data structure where each node satisfies the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key
- The right subtree of a node contains only nodes with keys greater than the node's key
- The left and right subtrees must also be binary search trees

Now it may appear that we've done a whole lot of work to make this fancy new data structure, but what benefit does it really give us? It has the same time complexities as the ordered list implementation, but it seems significantly more complex to implement.

Let's consider a few scenarios

1.

5

As discussed prior, a `List` is an ADT that represents a collection of elements that maintains an order. What are ways we can implement a `List`? What are the time complexities of each of these implementations? What are the pros and cons of each implementation? How can we use these implementations to implement other ADTs, e.g. a `Stack` or a `Queue`?

Subsection 7.1
AVL Trees

Section 8
Hash Tables

Comment For the future