University of Warwick
Department of Computer Science

CS126

Design of Information Structures

Cem Yilmaz
June 10, 2022

Contents

1 Analysis of Algorithms

1.1 Classification of a good algorithm

A good algorithm is one that optimises the following:

- Time
- Memory
- Network bandwidth
- Energy consumption

However, the main focus in this module will be the running time, in particular, this $O(x)$

1.2 Running Time

Running time of an algorithm typical grows with the input size. However, for different inputs of the same size the running time of an algorithm can vary. Then, for an input of fixed size n, we have different running times. We can categorise them as the following:

- Average case - the typical running time an algorithm requires and is often very difficult to determine
- Best case - what is the minimum running time of the algorithm and is generally not useful
- Worst case - upper bound on the running time, for any possible input and is more standard to analyse. Our focus is generally this.

1.3 Finding the running time

1.3.1 Experimental Analysis

The first method to use is experimental analysis. For this, we use computer and run simulations. For this, we write a program implementing the algorithm and run the program with inputs of varying size and composition, noting the time needed. However, there are limitations:

- It is necessary to implement the algorithm. Sometimes this can be impossible.
- Need to make sure that we have considered all kinds of inputs. This sometimes cannot be possible and otherwise it would not be indicative of running time
- Different algorithms may run different in different systems due to different features. It can be especially worse for different hardware.

1.3.2 Theoretical Analysis

The second method is theoretical analysis. For this, we use pen and paper. We use a high-level description of the algorithm instead of an implementation. We then characterise running time as a function of the input size n which takes in account all possible inputs. This would indeed allow us to evaluate the speed of an algorithm independent of the hardware or software environment. A good way to do high level description is to use pseudo-code. We also assume that the algorithm runs in an idealised machine. We assume simple memory hierarchy that is unbounded, infinite precision in arithmetic operations etc.

1.4 Random Access Machine (RAM) Model

It is a simple mode of computation with a singular CPU. It only executes a single program. It also has a bank of memory cells where each cell can hold arbitrarily large positive integers. Every cell gets assigned an ID that allows us to access the information in some *unit time*. In CPU, as learned from CS132, the CPU has the program stored inside. It is also connected to a program counter and registers $R_0, R_1, \ldots$ along with memory cells. It can do basic operations between two numbers stored in the registers, which include but are not limited to:

- Addition

- Comparison
- Fetch an element from the memory
- Write an element to the memory

However, one thing that categorises and defines RAM as what it is is the fact that *we can access any memory cell in unit times*.

1.5 Primitive Operations

Primitive operations are basic computations that are performed by an algorithm. These take constant time in the RAM model and we count the primitive operations that happen. The assumption is that the number of primitive operations are proportional to the actual running time. Some examples of primitive operations include evaluating an expression, assigning a value, calling a method, indexing into an array etc.

Example 1.1. Operations in a code

```
1       public static double arrayMax(double[] data) {
2     int n = data.length;
3     double currentMax = data[0];
4     for (int j = 1; j < n; j++)
5       if (data[j] > currentMax)
6         currentMax=data[j];
7     return currentMax;
8   }
```

Listing 1: Operations

In particular,

- Step 3 has 2 operations
- Step 4 has 2 operations
- Step 5 has $2n$ operations
- Step 6 has $2n - 2$ operations
- Step 7 has $2n - 2$ operations and finally
- Step 8 has 1 operation.

The code for $arrayMax$ has $6n+1$ as the worst case and $4n+3$ as the best case. Let $T(n)$ be the running time of arrayMax. Then,

$$a(4n + 3) \leq T(n) \leq a(6n + 1)$$

Where a is the time to execute a primitive operation.

1.6 Sorting algorithms

Indeed, the time complexity can be seen clearly in sorting algorithms for times.

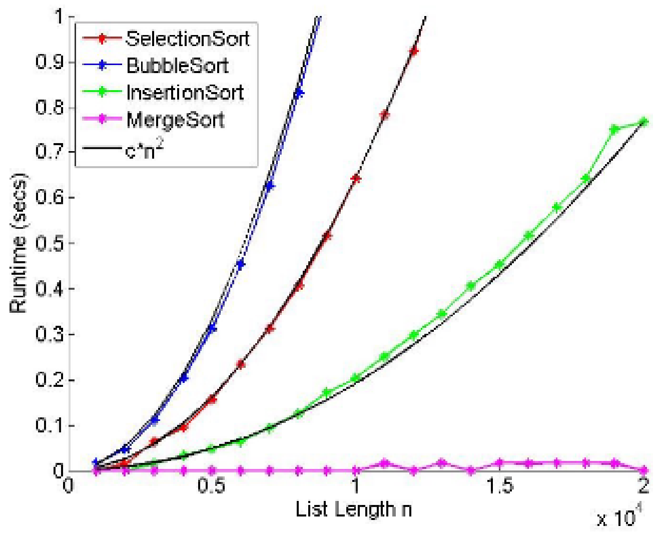


Figure 1: Complexity of sorting algorithms

2 Asymptotic Notation

2.1 Big-O notation

Definition 2.1. Big O

For two functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 i.e. $n_0, c \geq 1$ and such that

$$f(n) \leq cg(n) \tag{1}$$

for any $n \geq n_0$ and $n \geq n_0$.

Example 2.1. Example 1

$2n + 10$ is in $O(n)$.

$$2n + 10 \leq cn \tag{2}$$
$$(c-2)n \geq 10 \tag{3}$$
$$n \geq \frac{10}{c-2} \tag{4}$$

We pick $c = 3$ and $n_0 = 10$.

Example 2.2. Example 2

n^2 is not $O(n)$

$$n^2 \leq cn \tag{5}$$
$$n \leq c \tag{6}$$

c is a constant and thus this inequality cannot be satisfied.

Example 2.3. Example 3

$7n - 2$ is in $O(n)$

$$7n - 2 \leq cn \tag{7}$$

Pick $c = 7$ and $n_0 = 1$.

Example 2.4. Example 4

Is is true that $t > 0$, $(1+n)^t$ is in $O(n^t)$

$$(1+n)^t = \sum_{i=0}^{t} \binom{t}{i} n^i \tag{8}$$

However, t is the biggest number therefore it is $O(n^t)$

Thus, the Big-O notation gives an upper bound on the growth rate of a function as n grows towards infinity. We can use the Big-O notation to rank functions according to their growth rate.

2.1.1 General Rules

- Drop lower order terms
- Drop constant terms
- Use the smallest possible class of functions

2.2 Relatives of Big-O

2.2.1 Big Omega

Definition 2.2. Big Omega

$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $c > 0$ and an integer constant $n_0 \geq 1$ such that for $n \geq n_0$

$$f(n) \geq cg(n) \tag{9}$$

2.2.2 Big Theta

Definition 2.3. Big Theta

$f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and ann integer constant $n_0 \geq 1$ such that for $n \geq n_0$

$$c'g(n) < f(n) \leq c''g(n) \tag{10}$$

2.2.3 Summary

- Big-O is asymptotically less than or equal to $g(n)$
- Big-Ω is asymptotically greater than or equal to $g(n)$
- Big-Θ is asymptotically sandwiched between $g(n)$ differing by constant

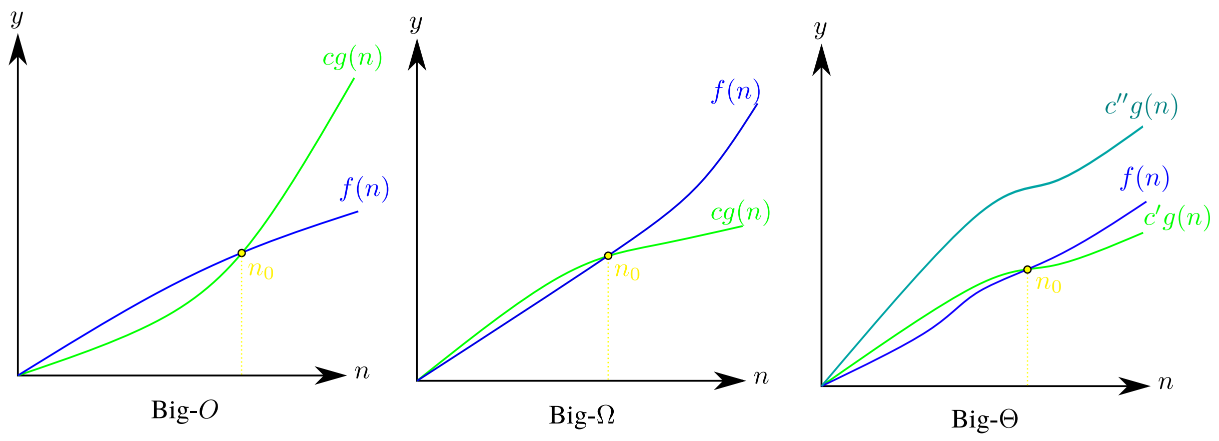


Figure 2: Summary of Big notations graphically

2.3 Examples

Example 2.5. Example 1

$5n^2$ is in $\Omega(n^2)$

$$5n^2 \geq cn^2, \text{ let } c = 5, n_0 = 1 \tag{11}$$

$5n^2$ is also $\Omega(n)$

$$5n^2 \geq cn, \text{ let } c = 1, n_0 = 1 \tag{12}$$

$5n^2$ is in $O(n^2)$

$$5n^2 < cn^2, \text{ let } c = 6, n_0 = 1 \tag{13}$$

Because $\Omega(n^2)$ and $O(n^2)$, we indeed can say that

$$\Theta(n^2) \tag{14}$$

Example 2.6. ln N!

Prove that $\ln N! = \Theta(N \ln N)$
There exists $a, b > 0$ and integer $N_0 > 0$ such that for any $N \geq N_0$ we have

$$aN \ln N \leq \ln N! \leq bN \ln N \tag{15}$$

Equivalently, it suffice to prove that

$$\ln N! = O(N \ln N)$$
$$\ln N! = \Omega(N \ln N)$$

We begin with proving the upper bound:

Proof.

$$\ln N! = \sum_{i=1}^{N} \ln(i)$$

We also have that ln is an increasing function, that is for $x > y$ we have $\ln(x) > \ln(y)$ and this implies that for any $1 \leq i \leq N$ we have that $\ln(i) \leq \ln(N)$. Hence, we substitute $\ln(i)$ with $\ln(N)$ in the equation above

$$\ln N! = \sum_{i=1}^{N} \ln(i) \leq N \ln(N)$$

Therefore indeed $\ln N! = O(N \ln N)$ □

We now prove the lower bound:

Proof.

$$\ln N! = \sum_{i=1}^{N} \ln(i) \geq \sum_{i=\lceil \frac{N}{2} \rceil}^{N} \ln(i)$$

We now observe that ln is an increasing function, that is for $x > y$ we have $\ln x > \ln y$ which implies that for any $\lceil \frac{N}{2} \rceil \leq i \leq N$ we have that $\ln i \geq \ln(\frac{N}{2})$. Let us now substitute $\ln(i)$ with $\ln(\frac{n}{2})$ in the equation above to obtain

$$\ln N! \geq \sum_{i=\lceil \frac{N}{2} \rceil}^{N} \ln(i) \geq \left\lceil \frac{N}{2} \right\rceil \ln(\frac{N}{2})$$

Which indeed shows that $\ln N! = \Omega(N \ln N)$. □

3 Abstract Data Types (ADT)

Definition 3.1. Abstract Data Type

An abstract data type is an abstract of a data structure. For an ADT we need to specify

- Data stored
- Operations on the data
- Error conditions associated with operations

In abstract data types on what each operation does, but we do not focus on how it does it. In Java, ADT is expressed usually by an interface.

3.1 The Stack ADT

Intuitively, we can think of a spring-loaded plate dispenser.

Definition 3.2. Stack

A stack is a collection of objects that are inserted and removed according to last-in, first out principle. In other words, "LIFO".

3.1.1 Fundamental Operations

- Push - puts an element at the top of the stack.
- Pop - gets the element from the top of the stack.

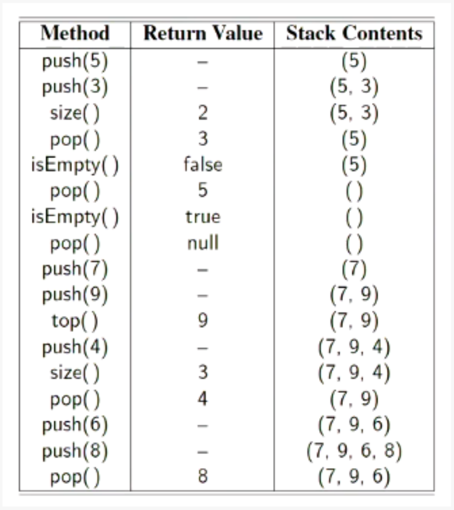
3.1.2 Main operations

- push(object): inserts an element
- pop(): removes and returns the last inserted element. Note that if it is empty we return null, an extreme case.
- we recall that insertions and deletions follow the LIFO scheme.

3.1.3 Auxiliary operations

- top(): returns the last inserted element without removing it
- size(): returns the number of elements stored
- isEmpty() indicates whether no elements are stored

Example 3.1. Operations of a stack



Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

Figure 3: Stack Operations

3.1.4 Application

The direction applications of stage include page visited history in a web browser, the undo sequence in a text editor and the chain of method calls in the Java Virtual Machine. Another distinct example is to use to match parentheses. The indirect applications include being auxiliary data structure for algorithms and component of other data structures. For example, we could use it to compute spans. For the array X, the span S is an array such that $S[i] =$ the number of consecutive elements $X[j]$ just before $X[i]$ such that $X[j] \leq X[i]$.

3.1.5 Implementation

The idea involves of adding elements from left to right of the array, with a variable that keeps track of the index of the top element. We would increment and decrement this variable as we push and pop objects into our array. If the array storing the elements suddenly becomes full, then a push operation will throw a FullStackException which is a limitation of the array based implementation. This is not intrinsic to the Stack ADT, and one could do a workaround using ArrayLists. The performance of array based limitation is as follows

- Space: It requires space $O(n)$ for storing n elements
- Time: Each operation runs in $O(1)$ time

Limitations include, but are not limited to

- The maximum size of the stack must be defined priori and can't be changed
- Trying to push a new element into a full stack causes an implementation-specific exception

In terms of performance, it is very good.

3.2 Queue ADT

Definition 3.3. Queue

A queue is a FIFO - first-in, first-out. In comparison to stack which is a LIFO. Insertions are at the rear of the queue and removals are the front of the queue. Priority is given to the one who arrived first. We generally tend to use queues for fairness. It stores arbitrary objects.

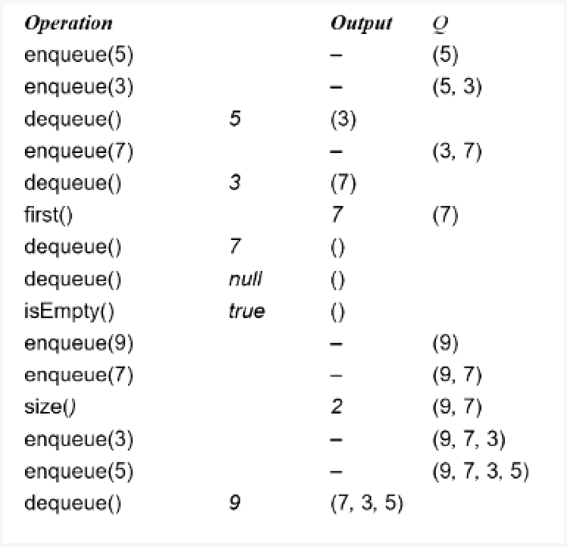
3.2.1 Main and fundamental operations

- enqueue(object): inserts the object at the end of the queue
- dequeue(): removes and returns the element at the front of the queue. Note that we return null if it is empty, an extreme case

3.2.2 Auxiliary operations

- first(): returns the element at the front without removing it
- size(): returns the number of elements stored
- isEmpty(): indicates whether no elements are stored

Example 3.2. Operations



Operation		*Output*	*Q*
enqueue(5)		–	(5)
enqueue(3)		–	(5, 3)
dequeue()	*5*	(3)	
enqueue(7)		–	(3, 7)
dequeue()	*3*	(7)	
first()		*7*	(7)
dequeue()	*7*	()	
dequeue()	*null*	()	
isEmpty()	*true*	()	
enqueue(9)		–	(9)
enqueue(7)		–	(9, 7)
size()		*2*	(9, 7)
enqueue(3)		–	(9, 7, 3)
enqueue(5)		–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)	

Figure 4: Operations of queue

3.2.3 Application

Direct applications include waiting lists - queuing theory in mathematics and access to shared resources e.g., printer. For example, round robin schedulers use queues.
Indirect applications include auxiliary data structures for algorithms and component of other data structures

3.2.4 Implementation

With arrays, the idea is to use an array of size N in a circular fashion. Two variables keep track of the front and the size. f index of the front element and sz number of stored elements. Note that it would be meaning to begin the queue in the middle of the array. When the queue has fewer than N elements, array location $r = (f = sz) \bmod N$ is the first empty slot past the rear of the queue. Note that because we want to be able to use all of the array in circles, it would be viable to implement the modulo N operator. In other words,

```
if isEmpty() then
  return null
else
  o = Q[f]
  f = (f+1) mod N
  sz = sz-1
  return o
```

Listing 2: dequeue

This is to ensure that overflow of queue can go to the beginning of array instead. In arrays, we cannot add more than N elements due to limitation of arrays. Furthermore, enqueue(o) should throw an exception if the array is full.

3.2.5 Performances and Limitations

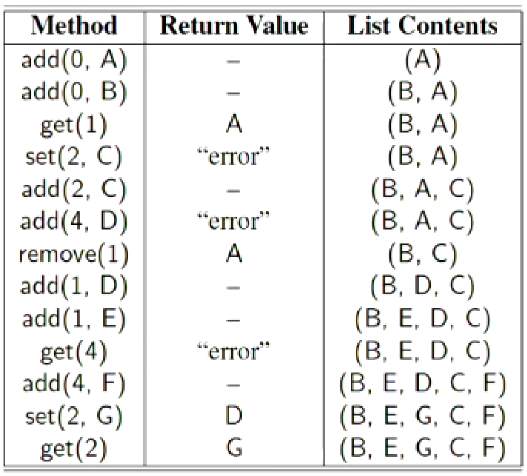
The performance space is that it requires $O(n)$ for storing n elements
The time required for each operation is $O(1)$ time
The limitations include the fact that the maximum size of the queue must be defined a priori and cannot be changed. Trying to enqueue a new element into a full queue causes an implementation-specific exception. In terms of performance, we could have not hoped for better as arrays give very quick access to elements.

3.3 List ADT

We have seen lists from data structures, namely section 4.2. These were data structures and not ADTs.

3.3.1 Main, fundamental and auxiliary operations

- set(i,e) - replaces the element at index i with e, and returns the old element that was replaced; an error occurs if i is not in range $[0, size() - 1]$
- get(i) - returns the element of the list having index i; an error condition occurs if i is not in range $[0, size() - 1]$
- size() - returns the number of elements in the list
- isEmpty() - returns a boolean indicating whether the list is empty
- add(i,e) - inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list. An error condition occurs if i is not in range $[0, size()]$.
- remove(i) - removes and returns the element at index i, moving all subsequence elements on index earlier in the list; an error condition occurs if i is not in the range $[0, size() - 1]$.



Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

Figure 5: List Operations Example

3.3.2 Implementation

A natural choice for implementing the list ADT is to use an array, A. The idea is that $A[i]$ stores a reference to the element with index i in the list. The methods get(i) and set(i,e) are easy to implement by accessing $A[i]$.
We insert element by using the operation add(i,o). The challenge is that we need to make room for the new element, meaning that we are shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$. The worst case corresponds to having $i = 0$, if there are n elements already in the list then we require to shift it by n. With removing elements, the challenge is to fill the hole left by the removed element. Shift backward the $n-i-1$ elements $A[i+1], \ldots, A[n-1]$, which means that at the worst we obtain n shifts.

3.3.3 Performance

Suppose that we have a list L with size n, there are n elements in L. The space required for L is $O(n)$, assuming that each reference object requires $O(1)$. Add and remove require in worst $O(n)$ operations. The limitations is that we cannot store more elements than the size of the array. If the array is full, the add operation throws an exception.

3.4 ArrayList ADT

In addition to the list ADT, If the array is full, instead of throwing an exception for push(o), let us

1. replace the array with a larger one
2. insert the new element

```
if size=S.length then
tmp = new array of size ...
for i = 0 to S.length-1
  tmp[i] = S[i]
S = tmp
S[n] = o
```

Listing 3: push

The question stands, how big should the new array be:

- Incremental strategy - increase the size by some constant c
- Doubling strategy - double the size

Let us measure the comparison of the two strategies. Consider $T(n)$ the elementary operations needed to perform a series of n push operations. Each one of the incremental and double strategy gives a rise to a different $T(n)$. We will compare the strategies in terms of corresponding quantities $T(n)$ Assume

- We start with an empty list
- The initial size of the array is 1

We call amortised time of a push operation the "average time" each push operation require i.e., $\frac{T(n)}{n}$.

3.4.1 Incremental strategy - analysis

Over n push operations we place the array $k = \lfloor \frac{n}{c} \rfloor$ times, where $c > 0$ is a constant independent of n. The total time $T(n)$ of a series of n push operations is proportional to

$$\begin{aligned} c + 2c + 3c + 4c + \ldots + kc &= \\ c(1 + 2 + 3 + \ldots + k) &= \\ \frac{c}{2}k(k+1) \end{aligned}$$

Note that the reason for increase is because we keep adding new elements and when creating the new array we require to create a bigger array and transfer more data each time, and this is incremented by c each time for an array of size c.
Since c is a constant, $T(n) = \Omega(k^2) = \Omega(n^2)$. The amortised time of a push operation is $\Omega(n)$. The above analysis gives us an "asymptotic lower bound" for the amortised time.

3.4.2 Doubling strategy - analysis

Over n pushes operations we replace the array $k = \log_2 n$ times. The total time $T(n)$ of a series of n push operations is proportional to

$$1 + 2 + 4 + 8 + \ldots + 2^k = 2^{k+1} - 1 = 2n - 1$$

Therefore $T(n)$ is $O(n)$, the amortised time of a push operation is then on average $O(1)$.

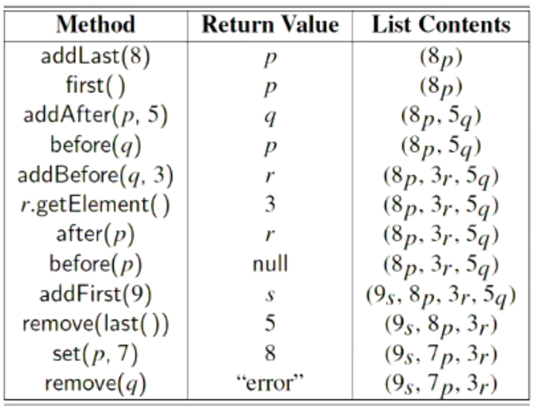
3.4.3 Remarks on strategies

Although amortised time for doubling strategy is only $O(1)$, we need to realise that the doubling strategy is not so economical in the use of space. With the doubling strategy we win in time/operation but we lose in space usage.

3.5 Positional List ADT

3.5.1 Main, fundamental and auxiliary operations

- first(): returns the position of the first element of L or null if empty
- last(): returns the position of the last element of L or null if empty
- before(p): returns the position of L immediately before position p (or null if p is the last position)
- after(p): returns the position of L immediately after position p (or null if p is the last position)
- isEmpty(): returns true if list L does not contain any elements
- size(): returns the number of elements in list L
- addFirst(e): Inserts a new element e at the front of the list, returning the position of the new element
- addLast(e): Inserts a new element e at the back of the list, returning the position of the new element
- addBefore(p,e): Inserts a new element e in the list, just before position p, returning the position of the new element
- addAfter(p,e): Inserts a new element e in the list, just after position p, returning the position of the new element
- set(p,e): replaces the element at the position p with element e, returning the element formerly position p.
- remove(p): removes and returns the element at position p in the list, invalidating the position.



Method	Return Value	List Contents
addLast(8)	p	(8_p)
first()	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement()	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last())	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

Figure 6: Positional List Operations

A natural way to implement positional list ADT is by the doubly linked list.

3.6 Maps ADT

Definition 3.4. Maps

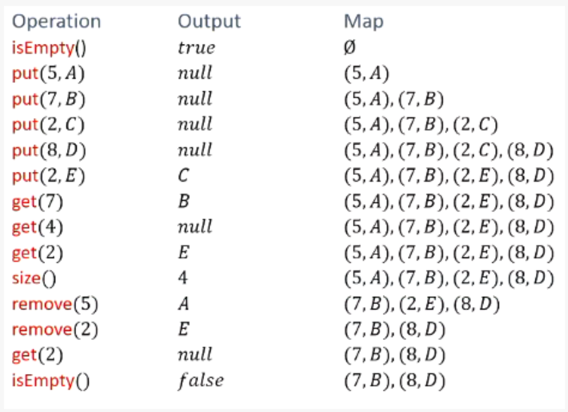
Maps are searchable collection of key-value entries where each entry is $(key, value)$

Intuitively, a map M supports the **abstraction** of using keys as indices with a syntax like $M[k]$, very similar to that of an array. As a map with n items uses keys that are known to be integers in a range from 0 to $N - 1$, for some $N \geq n$ we can just use an array..

3.6.1 Main, fundamental and auxiliary operations

- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k,v): insert entry (k, v) into the map M ; if the key k is not already in M, then return null; else, return old associated value with k.
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

Example 3.3. Operations example



Operation	Output	Map
isEmpty()	*true*	∅
put(5, *A*)	*null*	(5, *A*)
put(7, *B*)	*null*	(5, *A*), (7, *B*)
put(2, *C*)	*null*	(5, *A*), (7, *B*), (2, *C*)
put(8, *D*)	*null*	(5, *A*), (7, *B*), (2, *C*), (8, *D*)
put(2, *E*)	*C*	(5, *A*), (7, *B*), (2, *E*), (8, *D*)
get(7)	*B*	(5, *A*), (7, *B*), (2, *E*), (8, *D*)
get(4)	*null*	(5, *A*), (7, *B*), (2, *E*), (8, *D*)
get(2)	*E*	(5, *A*), (7, *B*), (2, *E*), (8, *D*)
size()	4	(5, *A*), (7, *B*), (2, *E*), (8, *D*)
remove(5)	*A*	(7, *B*), (2, *E*), (8, *D*)
remove(2)	*E*	(7, *B*), (8, *D*)
get(2)	*null*	(7, *B*), (8, *D*)
isEmpty()	*false*	(7, *B*), (8, *D*)

Figure 7: Map operations example

Note that multiple entries with the same key are not allowed.

3.6.2 Implementation

Our implementation will be done using an unsorted list this time. We store the items of the map in a list S (based on a doubly linked list) in an arbitrary order. For example, to implement get(k) we would

```
algorithm get(k):
B = S.positions()
while B.hasNext() do
  p = B.next()
  if p.element().getKey() == k then
    return p.element().getValue()
return null
```

Listing 4: get(k)

and for put

```
B = S.positions()
while B.hasNext() do
  p = B.next()
  if p.element().getKey() == k then
    t = p.element().getValue()
    S.set(p,(k,v))
    return t //return the old value
```

```
S.addLast((k,v))
size = size+1 // increase size as we are adding
return null //no previous entry was found
```

Listing 5: put(k,v)

and for remove

```
B = S.positions()
while B.hasNext() do
  p = B.next()
  if p.element().getKey() == k then
    t = p.element().getValue()
    S.remove(p)
    size--
    return t
return null
```

Listing 6: remove(k)

3.6.3 Applications

Address book and student-record databases.

3.6.4 Performance

- put requires $O(n)$ operations in the worst case as it may need to traverse the entire sequence to check whether the new element already exists.
- get and remove for similar reasons also require $O(n)$ times, as we require to search to see if the k exists.
- The performance in general is poor, we would like to have $O(\log n)$ or $O(1)$ operations, and the unsorted list implementation is effectively only for maps of small size. For example, sort would be able to give us a method of implementing binary search to search for the K.

3.6.5 Hashing

To generalise our previous question about having non-integer keys we can use something called the hashing function

Definition 3.5. Hashing Function

A hash function generates a map of data of arbitrary size a fix size value. This generates keys for any input key. It begins with hash code:

$$\begin{aligned} h_1 &: \text{keys} \to \text{integers} \\ h_2 &: \text{integrs} \to [0, N-1] \end{aligned}$$

The hash code is applied first and the compression function is applied next to the result

Example 3.4. Hash Function Example

The $h(x)$ defined as

$$h(x) = x \bmod N \tag{16}$$

is a hash function for integer keys

Definition 3.6. Hash Table

A has table for a given key type consists of

- hash function $h(x)$
- array, of size N called table

When implementing a map with hash table, the goal is to store item (k, o) at the index $i = h(k)$.

For example, entries with (SSN, Name), say we use an array data of length 10000 and store entry (k, v) in cell data[i] where $i =$ last 4 digits of k. Then,

$$h(025 - 612 - 0001) = 0001$$

Definition 3.7. Collision

Suppose for our hash function above for SSN, what happens if there are distinct datas that generate the same hash index? In such case, the answer is not trivial with no clear solution. Note that collision is bound to happen because by nature hashing is the idea of mapping an arbitrarily large size data to a fixed data size.

Definition 3.8. Horner's Rule

Polynomial $p(z)$ can be evaluated in $O(n)$ time

$$p_0(z) = a_{n-1} \tag{17}$$
$$p_i(z) = a_{n-i-1} + zp_{i-1}(z) \tag{18}$$

for $i = 1, 2, \dots, n-1$ we have $p(z) = p_{n-1}(z)$. That is, each new p is computed from the previously one in $O(1)$ time

3.6.6 Methods of reducing collision (Collision Handling)

- Separate chaining - let each cell in the table point to a map of entries, i.e., multiple instances in one cell. Separate chaining is simple, but requires additional memory outside the table
- Linear probing - reduce collision through open addressing. The colliding item is placed in a different cell of the table. It handles collisions by placing the colliding item in the next available table cell. This can cause lumps of collided elements
- Double hashing - uses two hash functions h and f. If cell $h(k)$ is already occupied, it tries sequentially another cell. The secondary has function $f(k)$ cannot have zero values. The table size N must be a prime to allowing probing of all the cells. It generalises linear probing. E.g., $h(k) + if(k) \bmod N$. In some sense, the secondary function allows for lumps not to be created because of the second function.

3.6.7 Performance

Worst case, searches, insertions and removals on a hash table take $O(N)$ time. The worst case scenario occurs when all the keys inserted into the map collide. In practice, we like to deal the hush function as generating random hash values. Worst case analysis is not the optimal approach. We measure this by load factor

$$\alpha = \frac{n}{N}$$

affects the performance of a hash table. n is the number of elements in table and N is the size of the table. Each has value we generate has probability α of creating collision. The "expected number" of problems for an insertion with open addressing is

$$\frac{1}{a - \alpha}$$

In practice, hashing is very fast provided the load factor is *not* close to 1.

3.7 Priority Queue ADT

3.7.1 Main, fundamental and auxiliary operators

A priority queue stores a collection of entries where each entry is a pair (key, value)

- insert(k,v)
- removeMin() removes and returns the entry with smallest key, or null if the priority queue is empty

Auxiliary methods include

- min()
- size()
- isEmpty()

3.7.2 Implementation

We can implement priority queue using an unsorted list. Insertion takes $O(1)$ time, removeMin and min takes $O(n)$ time since we have to traverse the entire sequence to find the smallest key.

3.8 Entry ADT

3.8.1 Main, fundamental and auxiliary operators

An entry in PQ a key-value pair. It stores entries to allow for efficient insertion and removal based on keys. Methods for entry include:

- getKey(): returns the key for the entry
- getValue: returns the value in this entry

3.9 Tree ADT

3.9.1 Main, fundamental and auxiliary operators

We use positions to abstract nodes. Generic methods:

- size()
- isEmpty()
- iterator()
- positions()

Accessors methods include

- root()
- parent()
- children(p)
- numChildren(p)

3.9.2 Traversal

Preorder traversal is a method of going through all nodes of a tree in a systematic manner. In a preorder traversal, a node is printed before its descendants

3.10 Set ADT

3.10.1 Main, fundamental and auxiliary operators

- add(e): Adds the element e to S if not already present
- remove(e): Removes the element e from S if present
- contains(e): Returns whether e is an element of S
- iterator(): Returns an iterator of elements of S

For which it is also usually implemented with traditional mathematical set operations i.e., $S \cap T$, $S \cup T$, $S - T$. Auxiliary methods include:

- addAll(T): Updates S to include all elements of set T.
- retainAll(T): Updates S so that it only keeps those elements that are also elements of the set T.
- removeAll(T): Updates S by removing any of its elements that also occur in the set T.

3.10.2 Implementation

We can implement a set with a list. Elements are stored according to some canonical ordering with space $O(n)$. Things such as union can be implemented using GenericMerge 6.2. For example, let

$$A = \{5, 7, 11, 12, 15\} B = \{3, 5, 14, 15, 22, 27\}$$

Then, our Generic Merge would be as follows:

1. Compare 5 and 3. Notice $3 < 5$, then, remove 3 from B and add it to S.
2. Compare 5 and 5. Notice $5 = 5$, then, remove both 5 from A and B and add it to S.
3. Compare 11 and 14. Notice $11 < 14$, then, remove 11 from A and add it to S.
4. ...
5. We then add the last elements of the bigger set.

Note that these run in linear time i.e., $O(n_A + n_B)$ time, provided the auxiliary methods run in $O(1)$ time.

4 Data Structures

4.1 Arrays

Definition 4.1. Array

An array is a sequences collection of variables of thesame type. Each variable, or cell, in an array has an index, which uniquely refers to the value stored in that cell.
A value stored in an array is often called an element.
The length of an array determines the maximum number of elements that can be stored.

4.1.1 Strengths

- We assume that we can access each cell k in $O(1)$ time.
- You can write or read once accessed.
- Access time is very fast $O(1)$.

4.1.2 Limitations

- We cannot change the length of an array

4.1.3 Declaring arrays in Java

Assignment to a literal form when initially declaring the array

```java
elementType[] arrayName = { v0, v1, ..., vn-1}
```

Listing 7: Array

elementType : any Java base type, or class name
arrayName : any valid Java identifier
Remark : the initial values must be of the same type as the array.
We also use the new operator to declare arrays because it is not an instance of a class. That is,

```java
new elementType[length]
```

Listing 8: Declaring array

The new operator returns a reference to the new array. This is assigned to the array variable measurements.

```java
double [] measurements = new double [1000]
```

Listing 9: Example

4.1.4 Examples

A array can store primitive elements, such as characters. E.g.

Table 1: Primitive element array

S	A	M	P	L	E
0	1	2	3	4	5

...or pointer to references to objects. For example, the first cell can refer to the pointer of the word "Joseph" and the second can refer to the pointer of "Helen" etc.

4.1.5 Adding an entry

To add an entry e into array board at i we need to make room. We shift each $n - i \mapsto n - i + 1$. This is $O(n - i)$.

4.1.6 Concluding Remarks

- Read/write any element in $O(1)$ time
- The capacity does not change
- Shifting k elements requires $O(k)$ time
- Very easy to work with
- We are going to use arrays a lot when we implement

4.2 Linked Lists

4.2.1 Singly Linked List

Definition 4.2. Singly Linked List

A singly linked list is a concrete data of structure consisting fro a sequence of nodes, starting from a head pointer.

Each node stores element and a link to the next node. However, nodes do not have information on previous nodes. The first node is usually called head, and the last is called tail. However, if the node is tail, then the next link is NULL. However, to access k-th element in the node, we would then require to repeat getting node $k-1$ times. This is called pointer hopping.

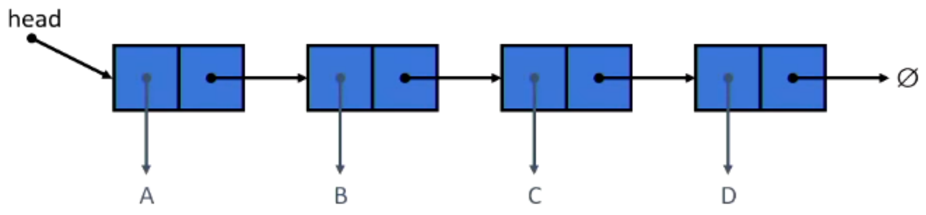


Figure 8: Singly Linked List

Note that the element after the last node is set to null.

4.2.2 Time required for Singly Linked List

Our assumption is that CurrentNode.getNext() requires $O(1)$ time. In order to access k-th element, we need $O(k)$ time.

4.2.3 Insertion of head

To insert a head:

- Allocate a new node
- Insert a new element
- New node points to old head
- Update head to point new node

This requires $O(1)$ time to insert a new head, as all of the steps taken are a constant.

4.2.4 Insertion of tail

To insert a tail:

- Allocate a new node
- Insert a new element
- New node points to null
- Have old last node point to new node
- Update tail to point to new node

This also requires $O(1)$ time, as all of the steps taken are a constant.

4.2.5 Remarks

There is no bound to the maximum number of elements we can store in a list. Therefore, as opposed to the array where the capacity is fixed, i.e., does not changed, the list is more flexible. Furthermore, the operation for insert is only $O(1)$ same as arrays. However, note that when inserting data in the middle of a linked list, we would require to search for the previous node which takes $O(k)$ time. The insertion itself, however, is $O(1)$.

4.2.6 Removing from head

Removing from the head requires

- Update head to point to next node in the list
- Garbage collector reclaims the former first node

This therefore requires $O(1)$ time.

4.2.7 Removing from tail

Removing from tail requires

- Update tail to null
- We need to access the node that shows towards the tail

This operation therefore requires $\Theta(k)$ time, making it an expensive operation as we required to gain access to the tail.

4.2.8 Doubly linked list

In a doubly linked list, nodes store an element, link to the previous node and a link to the next node.

4.2.9 Insertions in a doubly linked list

Similar to singly linked list, we would now also have to update pointers for previous pointers also.

4.2.10 Deletion in a doubly linked list

Deletion of a pointer in the trailer is now $O(1)$ since we can simply access it from the last node. Everything else stays the same.

4.2.11 List VS Array

Access time for k th element:

- Array requires $O(1)$
- Singly liked list requires time $O(k)$
- Doubly linked list requires time $min\{O(k), O(n-k)\}$, we can traverse it from the back

For capacity

- Array does not change capacity
- In both kinds of lists we can add and remove element with no restriction
- The time we need to add or remove an element in the list depends n the time we need to access the position of the list that we need to add or remove the element.
- Addition and removal is faster when ti is applied to the first or last element and takes $O(1)$ time.

5 Java Fundamentals

5.1 Recursion

5.1.1 Factorial

The recursive definition for factorial is:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{otherwise} \end{cases}$$

```java
public static int factorial(int n) throws IllegalArgumentException {
if (n < 0)
  throw new IllegalArgumentException();
else if (n == 0)
  return 1;
else
  return n * factorial(n-1);
}
```

Listing 10: Factorial in Java

5.1.2 Content of a recursive method

There is always a base case. For example, for factorial we have the base case as $n = 0$.
There is also always also recursive calls. Each recursive call should be defined so that it makes progress to a base case.

5.1.3 Linear recursion

Recur once - perform a single recursive call. This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls. The term "linear recursion" reflects the structure of the trace and not the asymptotic analysis.

5.1.4 Binary recursion

Binary recursion performs two recursive calls.

Example 5.1. Integer array A adder

```
Algorithm: $binarySum(A, i, n)$
Input: Array of integers $A$, where Integers $0 \leq i, n < A.length()$.
Output: Sum of the $n$ entries in $A$ starting at index $i$.
if $n = 1$ then
return $A[i]$
else return $binarySum(A, i, \lfloor \frac{n}{2} \rfloor) + binarySum(A, i + \lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{2} \rfloor)$
```

This in fact creates a binary tree.

Example 5.2. Fibonacci

Fibonacci numbers is an infinite sequence of integers $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$. There is a nice recursive definition for which

$$F_i = F_{i-1} + F_{i-2} \tag{19}$$

with boundary case corresponding to $F_0 = 0$ and $F_1 = 1$. Then, we cold write

```java
    if (n=1 || n=0)
      return n;
    else
      return binaryFib(n-1)+binaryFib(n-2)
```

Listing 11: Fibonacci

Everything in the operation is $O(1)$, however, let us denote n_k be the number of recursive calls by $BinaryFib(k)$. It seems that n_k doubles every other time. This means $n_k > 2^{\frac{k}{2}}$. The running time, is then, exponential in k. Exponential algorithms are generally not good. This can be instead turned into linear recursion

5.1.5 Multiple recursion

Here is a summation puzzle, where

1. $pot + pan = bib$
2. $dog + cat = pig$
3. $boy + girl = baby$

Our task is to assign a unique digit $0, 1, \ldots, 9$ to each letter in the equation in order to make the equation true. There are n^m solutions where n is the number of digits and m is the number of letters.

Example 5.3. Multiple recursion solution

Algorithm: $PuzzleSolve(k, S, U)$.
Input: integer$0 < k$, set of elements S, U
Output: enumeration of all k-element extensions to S using elements in U, without repetitions
For example, say $k = 1$ and $U = \{3, 4, 5\}$ and $S = (1, 2)$. Then the algorithm generates the sequences $(1, 2, 3), (1, 2, 4), (1, 2, 5)$ If we had $k = 2$, then we need to complete $(1, 2, *, *)$ using $\{3, 4, 5\}$ without repeating numbers. We can see that if we add k, the number of solutions double.

```
for each element e in U do
add e to the end of S
if k = 1, then
  test whether S is a configuation that solves the puzzle
  if S solve sthe puzzle then
  return "Solution found:" S
else
  PuzzleSolve(k-1,S,U)
add e back to U
remove e from the end of S
```

Listing 12: PuzzleSolve

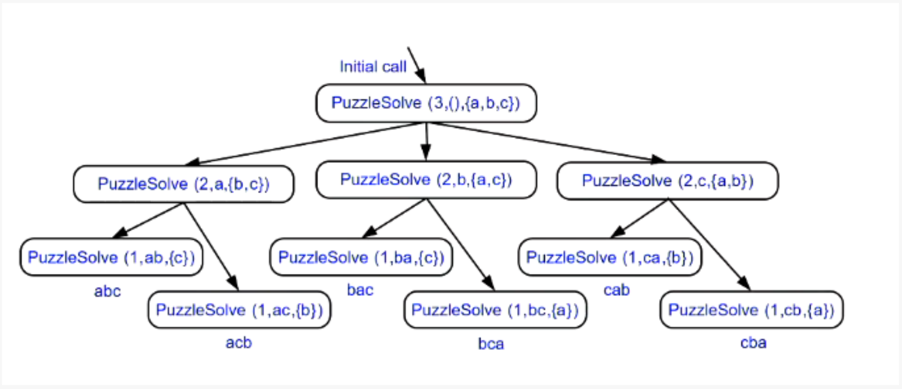


Figure 9: Puzzle Solver Visualisation

5.2 Iterators

Sometimes for abstract data structures we require to iterate over elements of an array. For example, if we want to copy array A onto some array B, then we would

```
for (i=0; i<A.length; i++) {
  B[i] = A[i]
}
```

Listing 13: Array Copy

Definition 5.1. Iterator

Iterators are software design pattern which abstracts the process of scanning through a sequence one element at a time. A good example is a for-each loop, which is an implicit iterator.

However, this does not allow us to iterate over the elements of a positional list. For this we use a construction called iterator. It is common in OOP languages. We would write this as

```java
Iterator<String> iter;
while (iter.hasNext()){
  String value = iter.next();
  System.out.println(value);
}
```

Listing 14: Iterator

The commands

- hasNext(): returns true if there is at least one additional element in the sequence and false otherwise
- next(): Returns the next element in the sequence

Java defines a parameterised interface named Iterable. Iterator() returns an iterator of the elements in the collection. An instance of a typical collection class in Java, such as an ArrayList, is iterable. It produces an iterator for its collection as the return value of iterator(). Each call to iterator() returns a new iterator instance. It also allows for multiple and simultaneous traversals of a collection. for each loops iterate over the element by the syntax

```java
for (ElementType variable : collection) {
  loopBody
}
```

Listing 15: For each loop

6 Sorting

6.1 Sorting Problem

We are given a sequence of unordered elements, for example

$$[3, 6, 2, 7, 8, 10, 22, 9]$$

We want to put the elements into a non decreasing order, i.e., from smaller to larger, in particular we want

$$[2, 3, 6, 7, 8, 9, 10, 22]$$

6.2 Generic Merge

Generalised merge of two sorted lists A and B. We will use the template method $genericMerge$ with auxiliary methods aIsLess, bIsLess and bothAreEqual.

```
algorithm genericMerge(A,B)
  S <- empty sequence
  while not(A.isEmpty) and not(B.isEmpty())
    a <- A.first().element(); b <- B.first().element()
    if a < b
      aIsLess(a,S); A.remove(A.first())
  2222222222 else if b < a
      bIsLess (b,S); B.remove(B.first())
    else
      bothAreEqual(a,b,S)
      A.remove(A.first()); B.remove(B.first())
```

```
while not(A.isEmpty())
  aIsLess(a,S); A.remove(A.first())
while not(B.isEmpty())
  bIsLess(b,S); B.remove(B.first())
return S
```

Listing 16: Psuedo code for generic merge

6.3 Divide and Conquer

Divide and conquer is a general algorithm design paradigm that:

- Divide: Divide the input S into two disjoint subsets S_1 and S_2.
- Recur: Solve the sub problems associated with S_1 and S_2.
- Conquer: Combine the solutions for S_1 and S_2 into a solution for S.

The base case for the recursion are sub problems of size 0 or 1.

6.4 Merge Sort

6.4.1 Method

Merge sort utilises divide and conquer. The input is a sequence S with n elements. There are 3 steps:

1. Divide: partition S into two sequences S_1 and S_2 of about $\frac{n}{2}$ elements each
2. Recur: recursively sort S_1 and S_2
3. Conquer: merge S_1 and S_2 into a unique sorted sequence

```
input: sequence S of n elements
output: sequence S sorted
if S.size() > 1
  (S1,S2) <- partition(S, n/2)
  mergeSort(S1)
  mergeSort(S2)
  S <- merge(S1,S2)
```

Listing 17: mergeSort(S)

The clever part of this sort comes from its merge function, which is defined as the generic merge 6.2. Note that merging two sorted sequences, each with $\frac{n}{2}$ elements and implemented by means of a doubly linked list, takes $O(n)$ time.

6.4.2 Running time

The worst case amount of comparisons is $O(n_A + n_B)$ when one of the lists size is not the same as the other and the best case is $min(n_A, n_B)$. The best case occurs when one of the sets is strictly bigger/smaller than every element in the other set.
The height h of merge-sort tree is $O(\log n)$

Proof. The overall amount of work done at the nodes of depth i is $O(n)$. We partition and merge 2^i and sequences of size $\frac{n}{2^i}$. We make 2^{i+1} recursive calls. Thus the total running time of merge-sort is $O(n \log n)$. In otherwords, $O(\log n) \times O(n) = O(n \log n)$. That is, it is the work done at each depth of node times the height.
Remember that the last depth of the tree is actually the height h. Then, we know for the number of nodes n

$$1 + 2 + 2^2 + \ldots + 2^{h-1} = \frac{1-2^h}{-1} \text{ Geometric Sequence formula}$$
$$n = 2^h - 1$$
$$\log_2(n+1) = h$$

□

6.5 Sorting with Priority Queue

1. Insert the elements one by one into the PQ with a series of insert()
2. Remove the elements in sorted order with a series of removeMin()

The running time of this sorting method depends on the priority queue implementation. For selection sort, we use the variation of PQ sort where the priority queue uses an unsorted sequence. Selection sort takes

$$O(n) + O(n-1) + \ldots + O(1) = O(n^2)$$

Selection sort takes $O(n^2)$. Similarly, there is insertion sort. The variation of PQ sort where the priority queue is implemented with a sorted sequence, the running time of insert-sort is $O(1) + O(2) + \ldots O(n) = O(n^2)$ due to insertion. removeMin then would take $O(n)$.

6.6 Quicksort

6.6.1 Method

Quicksort is a randomised sorting algorithm based on the divide-and-conquer paradigm:

1. Divide: pick a random element x (called pivot) and partition S into
   - L elements less than x
   - E elements equal x
   - G elements greater than x
2. Recur: sort L and G
3. Conquer: join L, E and G

```
input: sequence S. position p of pivot
output: subsequences L,E,G of the elemenets of S less than, equal to, or greater than the pivor
     respectively
L,E,G <- empty sequences
x <- S.remove(p)
while not(S.isEmpty())
  y <- S.remove(S.first())
  if y < x
    L.addLast(y)
  else if y == x
    E.addLast(y)
  else
    G.addLast(y)
  return L,E,G
```

Listing 18: Partition

We partition an input sequence S by removing in turn each element y from S and insert y into L, E or G, depending on the result of the comparison with the pivot x.

6.6.2 Running time

Each insertion, removal and comparison takes $O(1)$ and thus the partition step of quick-sort takes $O(n)$ time. Worst case for quick-sort occurs when the pivot is the unique minimum or the unique maximum element. One of L and G has size $n-1$ and the other has size 0. The running time is proportional to the sum, then,

$$n + (n-1) + \ldots + 2 + 1 = \qquad O(n^2)$$

The expected running time, that is good call, where the sizes of L and G are each less than $\frac{3s}{4}$. A bad call one of L and G has size greater than $\frac{3s}{4}$. The expected height of quicksort is $O(\log n)$. The amount of work done is at the nodes of the same depth is $O(n)$. Therefore the expected running time is $O(n \log n)$.

7 Search

7.1 Binary search

7.1.1 Introduction

This is used to efficiently locate a "target value" within a **sorted** sequence of n elements. For example,

Table 2: Binary search

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

The trivial thing to do is to scan the sequence from left to right. We need $O(n)$ time worst-case. If the sequence is unsorted all the following do not apply... We would require to check everything which would then take $O(n)$ time.

7.1.2 Example

Example 7.1. Find 22

An element of the array is called a **candidate** if at the current stage we cannot rule out that this item matches the target value. Let us say that candidate is 22.

Table 3: Binary search

;

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Note that 2 $=$ low, 14 $=$ mid and 37 $=$ high. There are now 3 cases to consider:

1. 22 $=$ mid
2. 22 $<$ mid
3. 22 $>$ mid

Then, for case 1, we have found the target element and we stop the algorithm. For case 2, algorithm recurs on the first half of the array i.e., our new high = mid. Similarly, for case 3, we have set that low = mid. Note that if it was unsorted we would not be able to conclude these.

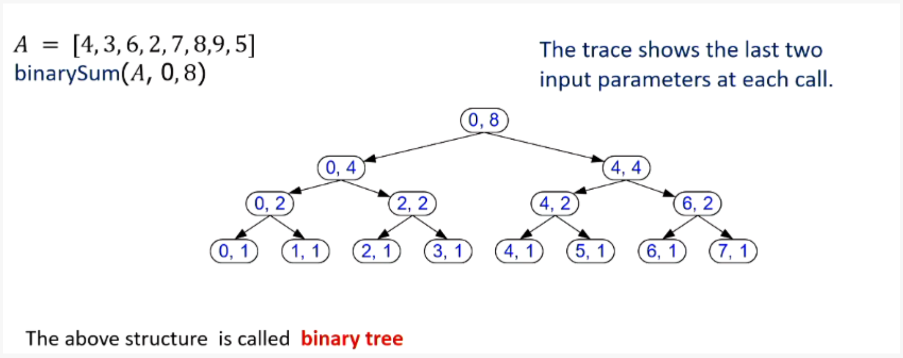


Figure 10: Binary Tree

7.1.3 Analysis

The input is an array of size k. The time is proportional to the number of recursive calls as accessing elements and comparing are all of $O(1)$. The number of recursive calls, however, is running in $O(\log n)$ times. The remaining portion of the list of size is $\text{high} - \text{low} + 1$. After one comparison, the number of candidate halves:

$$(\text{mid} - 1) - \text{low} + 1 = \left\lfloor \frac{\text{low} + \text{ high}}{2} \right\rfloor - \text{low} \leq \frac{\text{high} - \text{low} + 1}{2}$$
$$\text{high} - (\text{mid} + 1) + 1 = \text{high} - \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor \leq \frac{\text{high} - \text{low} + 1}{2}$$

After j th call, the number of candidates left is at most $\frac{n}{2^j}$. We need to have $\frac{n}{2^j} > 1$ to have the algorithm running.

8 Heap

A heap is a binary tree storing keys at its nodes and satisfying the following priorities:

- Heap-order: for every internal node v other than root, $key(v) \geq key(parent(v))$.
- Complete binary tree: let h be the height of the heap.
- The last node of a heap is the rightmost node of maximum depth.

A heap storing n keys has height $O(\log n)$. For a heap based implementation of Priority Queue the idea is that we store a (key,value) elements at each node following that smaller key is higher priority. For insertion, let k be the key element we insert to the heap. The insertion algorithm consists of three steps

- Insert a new node z (the new last node)
- Store k at z.
- Restore the heap-order property

To restore, we use upheap

- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- The algorithm terminates when it is either at the root or a node whose parent has a key smaller than k

The running time for upheap is $O(\log n)$ due to its height. Removal, however, gets slightly tricky. Recall removeMin() of the priority queue ADT. This means that removal of the root key from the heap.

- Replace the root key with the key of last node w.
- We remove w
- We restore the heap property using downheap

To restore, we use downheap

- The algorithm swaps the parent node with it's children which is the smallest.

With this, we bring heap sort

- Sequence S, of n elements
- Use PQ Sorting
- First phase we insert, one by one, the element in S to a priority queue P
- Second phase we extract the elements from P by a series of removeMin()
- Put them back into S
- Now use a heap-based implementation for the priority queue

The performance of this is $O(n \log n)$.

9 Skip-list

Slip list is a set $S = \{S_0, S_1, \ldots, S_h\}$. S_0 contains all elements, including $\pm\infty$. S_i is a random subset of S_{i-1} appear in S_i with probability $\frac{1}{2}$, independently. S_h contains only $\pm\infty$. To search, we start at the position of the top list. At the current position p, we compare x with $y \leq key(next(p))$.

1. $x = y$ we return element(next(p))
2. $x > y$ we scan forward
3. $x < y$ we drop down

For insertion of an element k, we search for k and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest element $< k$ in each list $S_0, S_1, \ldots, S_i$. We work as in the search algorithm. To determine where we add, we toss a coin. Skip lists are implemented with quad nodes. I.e., it stores previous, next, above and below data. The issue with skip lists is that space usage is high. The space used by skip list depends on the random bits used by each invocation of the insertion algorithm. In our coin tossing, the probability of getting i consecutive heads is

$$\left(\frac{1}{2^i}\right)$$

If each n entries is present a set with probability p_i the expected size of the set is np.

$$\sum_{j=0}^{\infty} \frac{n}{2^j} = 2n$$

The expected space usage of a skip list with n items is $O(n)$. The height with high probability, a skip list with n items has height $O(\log n)$. The search time and update time is proportional to number of drop-down steps and the number of scan-forward steps. The drop steps are bounded by height h of the skip list, i.e., $O(\log n)$ with high probability. The expected number of scan-forward steps is $O(\log n)$. Therefore it is $O(\log n)$.

10 Binary Search Tree

A binary tree storing keys at its internal nodes. External nodes do not store items. Property is for each internal p with key $k(p)$, the key in the left subtree $< k(p)$. The right is $> k(p)$. Usually

- $k(root) = k$ we found the element
- $k(root) < k$ k must be on the right
- $k(root) > k$ k must be on the left

If we reach a leaf, the key is not found.

10.1 Performance

Consider a binary tree with n items of height h. The space used is $O(n)$. Methods search,insert and remove take $O(h)$. The height h is worst case $O(n)$. Best case is $O(\log n)$.

10.2 AVL Tree

AVL trees are balanced regardless of the order we insert-delete elements of the tree. AVL tree is a binary search tree and for every internal node v of T, the height of the children of v can differ by at most 1. The above is also known as the height balance property. The height of an AVL tree storing n keys is $O(\log n)$

Proof. Let us bound $n(h)$: the minimum number of internal nodes of an AVL tree of height h. We can see that $n(1) = 1$ and $n(2) = 2$. For $n > 2$, an AVL tree of height h contains the root node, one AVL subtree of height $h-1$ and another of height $h-2$. That is, $n_9h) = 1 + n(h-1) + n(h-2)$. Knowing $n(h-1) > n(h-2)$ we get $n(h) > 2n(h-2)$. This implies $n(h) > 2^{\frac{h}{2}} > 1$. Taking logarithms we obtain $h < 2\log_n(h) + 2$
Insertion is as in binary search tree. Always done by expanding an external node. We call the balancing Trinode restructing. Let (a, b, c) be the inorder listing of x, y, z. Perform the rotations needed to make b the topmost node of the tree. □