Arrays - CS126

What is an array?

Sequenced collection of variables all of the same type.

Each cell has an index. ordering starts from 0

Declaring arrays

1. literally

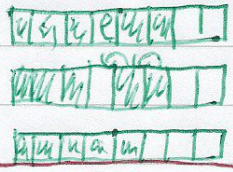
```
int[] primes = {2,3,5,7,11};
```

2. New keyword

```
int[] nums = new int[5];
```

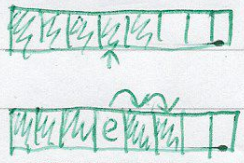
Removing

remove element at index and shift all elements right of that element once left



Adding

Need to find where to insert. Move all elements from there one to the right and insert new element



Insertion Sort - CS126

Algorithm

```
InsertionSort(A)
  for k from 1 to n-1 do
    Insert A[k] into correct position
```

Time complexity

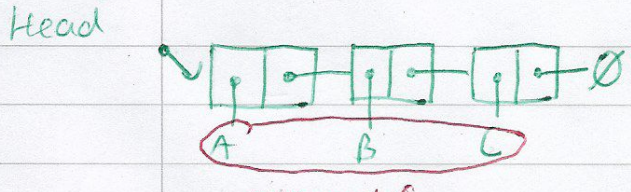
$O(n^2)$

$\Omega(n)$

Singly Linked Lists - CS126

What is a SLL

DS consisting of a sequence of nodes, starting from a head pointer



Insert - At head

Create new node point to old head, update head to point at new node

Insert - At tail

Create new node pointing to ∅

Set current tail node to point at new node.

Remove - from head

update head to point at next node

Remove - from tail

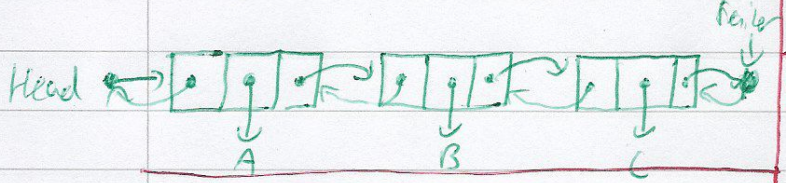
removing from tail is very inefficient.

No constant-time way to do this.

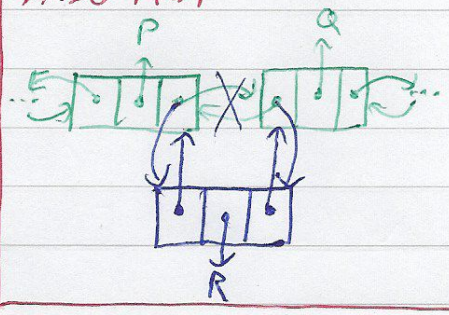
Doubly Linked Lists - CS126

What is a DLL?

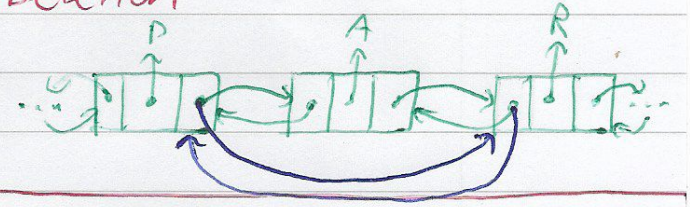
A linked list that can be traversed both ways.
Has a next and previous node.



Insertion



Deletion



Analysis of Algorithms - CS126

Running time:

Running time of algorithm tends to increase as input size increases.
We tend to focus on the worst case scenario

Theoretical Analysis

- uses high-level description of the algorithm
- Characterizes running time in Big-Oh notation.

Primitive Operations

- Basic computations performed by an Algorithm.
- Take constant time in RAM
- Number of primitive operations is proportional to the actual running time.

Experimental Studies

Write a program to implement the algorithm
Run the program with varying input sizes
Plot results.

Limitations

May be hard to implement
To compare algorithms they must be run in the same environment

Random-Access Machine model (RAM)

- one CPU
- executes a single program
- Infinite memory of infinite size.
- Can access all memory in unit time

We count the number of operations when calculating running time.

Analysis of Algorithms - CS126

8 important functions

Constant – 1 (Fastest)
Logarithmic – $\log n$
Linear – n
N-Log-N – $n \log n$
Quadratic – n^2
Cubic – n^3
Exponential – 2^n
Factorial – $n!$ (Slowest)

Big-Oh notation

Given $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 s.t.

$f(n) \leq c\,g(n)$ for $n \geq n_0$

Asymptotic time analysis

To perform this we:
- Find worst-case number of primitive operations.
- Express this in Big-Oh notation.

Big-Omega

$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an int $n_0 \geq 1$ s.t.

$f(n) \geq c\,g(n)$ for $n \geq n_0$

Big-theta

$f(n)$ is $\Theta(g(n))$ if there are constants $c', c'' > 0$ and an int $n_0 \geq 1$ s.t.

$c'g(n) \leq f(n) \leq c''g(n)$ for $n \geq n_0$

Growth Rate of running time

Changing environment:
- Affects running time by a constant factor
- Does not alter the growth rate.

Constant factors

Growth rate is not affected by:
- constants
- lower-order terms

Big-Oh and growth rate.

Big-Oh gives an upper bound on the growth rate of a function as $n \to \infty$.

Statement "$f(n)$ is $O(g(n))$" means the growth rate of $f(n)$ is no more than $g(n)$.

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	✓	✗
$f(n)$ grows more	✗	✓
grow the same	✓	✓

Intuition for Asymptotic notation.

O – $f(n)$ is ~~less~~ $\leq g(n)$
worst-case scenario

Ω – $f(n)$ is $\geq g(n)$
best-case scenario

Θ – $f(n) = g(n)$
Average-case scenario

Recursive Algorithms - CS126

Content of a recursive method

Base Case(s)

- Values of the input which cause us to not perform a recursive call
- Every chain of recursive calls must reach a base case

Recursive calls

- Calls the current method
- Each recursive call should make progress towards a base case

Linear Recursion

Performs a single recursive call.

Binary Recursion

2 recursive calls for each non-base case.

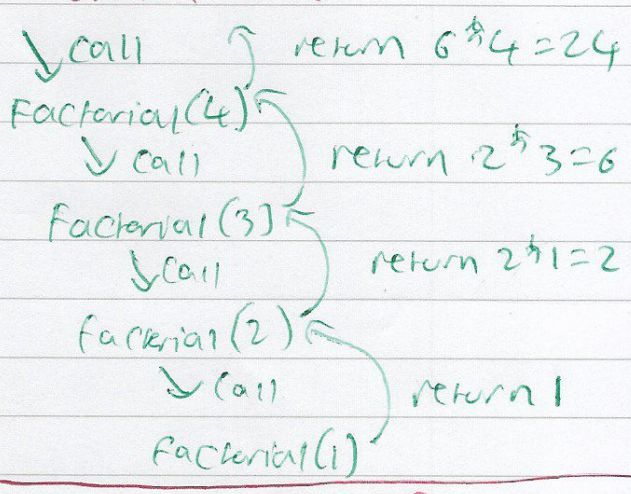
e.g. Fib

$f(0) = 1$ $O(2^n)$

$f(1) = 1$

$f(n) = f(n-1) + f(n-2)$

Recursion trace



Binary search - O(log n)

```
binSearch(arr, torset, low, high)
  if(low > high)
     return false
  mid = (low+high) / 2
  if(torset == arr[mid])
     return true
  if(torset < arr[mid])
     return binSearch(arr, torset, low, mid-1)
  return binSearch(arr, torset, mid+1, high)
```

Multiple recursion

Makes many recursive calls not just 1 or 2

Stack ADT - CS126

Abstract Data Type (ADT)

An abstraction of a Data Structure

Specifies:

- Data Stored
- Operations on the data
- Error conditions associated with operations.

In Java, an interface represents an ADT

Null instead of Exceptions

we could throw an exception if trying to pop from an empty Stack.

Or even better we can return NULL.

Method Stack - JVM

JVM keeps track of the chain of active methods

when a method is called JVM pushes a frame onto the Stack containing:

- Local variables & return value
- Program counter.

When a method ends it is popped from the Stack.

Allows recursion.

Limitations of array-based Stack

maximum size of Stack must be defined prior - cannot be changed.

Stack

Collection of objects inserted and removed according to LIFO principle.

Fundamental operations:

- Push
- Pop

Auxiliary operations

- top
- size
- isEmpty

Applications of Stacks

- Undo sequence in text editor.
- Chain of method calls in the JVM.
- Component in other DS's

Array-based Stack - O(n) space, O(1) time

Add elements left to right in the array

```
size()
  return t+1
```

```
pop()
  if isEmpty()
    return NULL
  t--=1
  return S[t+1]
```

```
push(o)
  if t = S.length - 1 then
    throw IllegalStateException
  else
    t++=1
    S[t]=o
```

Queues ADT - CS126

The Queue

Insert and remove following the FIFO principle.

Main queue operations:
- enqueue
- dequeue

Auxiliary operations:
- first
- size
- isEmpty

Boundary cases
- return null if dequeuing from empty queue

Applications of Queues

Waiting lists

Access to shared resources

Component of other DS's

Array based queue

Array of size N used in circular fashion.

2 variables used:
- f - front of queue
- sz - size of queue

Next empty slot calculated by:

$r = (f + sz) \bmod N$

Queue operations

```
size()                    isEmpty()
 return sz                 return (sz == 0)
```

```
enqueue(o)
 if (size() = N)
   throw IllegalStateException
 else
   r = (f + sz) mod N
   Q[r] = o
   sz += 1
```

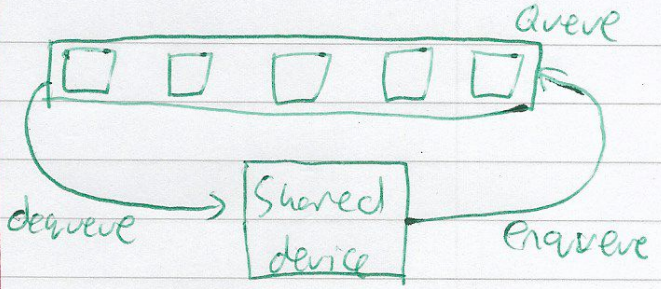
```
dequeue()
 if isEmpty()
   return null
 else
   o = Q[f]
   f = (f + 1) mod N
   sz -= 1
   return o
```

Round Robin Schedulers

CPU has limited capabilities of running simultaneous processes

Round Robin scheduler is an algorithm for different processes sharing a CPU

Can use a queue to do this



Lists and Iterators - CS126

List ADT
- size()
- isEmpty()
- get(i)
- set(i, e)
- add(i, e)
- remove(i)

Array Lists
A[i] is a reference to an element at index i.

Easy to implement get(i) and set(i, e).

Insertion
When adding we need to make room for the new element by shifting forward elements A[i], ..., A[n-1]

Worst case O(n).

Removal
We need to fill the whole left from removing. We do this by shifting elements A[i+1], ..., A[n-1] backwards by 1.

Worst case O(n)

Array based lists - performance
Space complexity - O(n)

Accessing elements by index - O(1).

add and remove - O(n)

Adding to ~~overscrown~~ full list results in an exception

Growable array
When the array becomes full we increase the size by replacing the array.

How large should the new array be?

1. Incremental Strategy:
   - increase by a constant C.
2. Doubling Strategy:
   - double the size

Incremental Strategy analysis
Over n operations we replace the array $k = n/c$ times.

Total time T(n) of a series of n push operations is proportional to:

$$c + 2c + 3c + 4c + \ldots + kc = c(1 + 2 + 3 + \ldots + k) = ck(k+1)/2$$

Thus, the amortized time of a push operation is $\Omega(n^*)$

Doubling Strategy analysis
replace array $k = \log_2 n$ times

Total time T(n) of a series of n push operations is proportional to:

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

Thus amortized time of a push operation is

Positional Lists
Abstraction of a sequence of elements with the ability to identify the location of an element, without indices.

A position acts as a marker or token within the broader positional list

Lists and Iterators 2 - CS126

Move on Positional Lists

A position p is unaffected by changes elsewhere in a list.

A position instance is an object.

DLL as positional list

A Doubly linked list can act as a positional list.

Iterable interface

Java defines a parameterized interface named Iterable.

- iterator(): Returns an iterator of the elements in the Collection.

Each call to iterator() returns a new iterator instance.

positional list ADT

- first()
- last()
- before(p)
- after(p)
- isEmpty()
- size()
- addFirst(e)
- addLast(e)
- addBefore(p,e)
- addAfter(p,e)
- set(p,e)
- remove(p)

Iterators

Abstracts the process of scanning through a sequence.

- hasNext()
- next()

```
while (iter.hasNext()){
    Object val = iter.next();
    do something;
}
```

Maps - CS 126

What is a Map?

Searchable collection of key-value pairs.

Main operations:
- Searching
- inserting
- deleting

Can not have 2 entries with the same key.

Map ADT

- get(k)
- put(k, v)
- remove(k)
- size()
- isEmpty()
- entrySet() - iterable
- keySet() - iterator
- values() - iterator

List-based Map

Store key-value pairs in a DLL, in arbitrary order.

Put analysis:

$O(n)$

get and remove analysis

$O(n)$

Only effective for maps of small size.

get(k)

```
Alg get(k)
  B = S.positions
  while B.hasNext()
    P = B.next()
    if P.element().getKey() = k
      return P.element().getValue
  return null
```

remove(k)

```
Alg remove(k)
  B = S.positions()
  while B.hasNext()
    p = B.next()
    if p.element.getKey() = k
      t = p.element.getValue()
      S.remove(p)
      n = n-1
      return t
  return null
```

put(k, v)

```
Alg put(k, v)
  B = S.positions
  while B.hasNext()
    p = B.next()
    if P.element().getKey() = k
      t = p.element().getValue()
      S.set(P, (k, v))
      return t.
  S.addLast((k, v))
  n = n+1
  return null
```

Hash Tables - CS126

Hash Tables as Maps

A Hash Table acts as a map but uses a method to turn any several keys into an index for an array.

Hash Table

Consists of:

- Hash function h
- Array of size N

Hash Functions

has 2 Components:

1. Hash Code
   h_1: keys → integers
2. Compression function
   h_2: integers → $[0, N-1]$

$h = h_2 \circ h_1$: keys → $[0, N-1]$

Compression Functions

Division

- $h_2(y) = y \bmod N$
- N is size of hash table and tends to be prime.

Multiply, Add and Divide (MAD)

- $h_2(y) = (ay + b) \bmod N$
- a and b are non-negative and $a \bmod N \neq 0$

Seperate Chaining

Used to fix collisions

If we map to an index that contains a pair we move this new pair to the end of the linked list.

Hash Function

used to convert a key into an index.

Many ~~one~~ different hash functions to use e.g.

last 4 digits

14235**6789** → A[6789]

Collisions

Collisions occur when 2 different keys get mapped to the same location.

Hash Codes

Memory address

reinterpret the memory address of the key as an integer

Integer Cast

- reinterpret bits as integer

Component Sum

partition bits of key into components of fixed length and sum components, ignoring overflow.

Polynomial accumulation

Partition bits into a sequence of components of fixed length.

$a_0, a_1, a_2, \ldots, a_{n-1}$

Evaluate polynomial

$P(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$

Can be evaluated in $O(n)$ time using Horner's rule

Hash Tables 2 - CS126

Seperate Chaining Functions

Each cell contains a linked list

```
get(k)
  return A[h(k)].get(k)
```

```
put(k,v)
  t=A[h(k)].put(k,v)
  if t=null
    n++
  return t.
```

```
remove(k)
  t=A[h(k)].remove(k)
  if t≠null
    n--
  return t.
```

Linear probing

handles collisions by placing the colliding item in the next available table cell

Removing with linear probing

Search for element to remove and replace with a DEFUNCT value

This ensures we don't lose save values.

Search with linear probing

```
get(k)
  i=h(k)
  p=0
  repeat
    c=A[i]
    if c=∅
      return null
    elif c.getKey()=k
      return c.getValue()
    else
      i=i+1 mod N
      p=p+1
  until p=N
  return null
```

Double Hashing

Uses 2 hash functions: h, f

If h(k) is occupied we try $(h(k)+i\cdot f(k)) \bmod N$ for $i=1,2,3\ldots$

- f can not result in 0.
- table size must be prime to allow probing of all cells.
- Common choice of f.
  $f(k)=q-k \bmod q$
  where
  $q<N$
  q is prime.

Performance of Hashing

- Worst case for search, insert and delete is $O(n)$.
- worst case is when keys collide
- load factor $\alpha = n/N$
- expected number of probes for an insertion with open addressing is $\frac{1}{1-\alpha}$
- In practice as long as α isn't near 1 Hash maps are very fast.

Sets - CS126

Set definition

an unordered collection of elements without duplicates that typically supports efficient membership tests

Storing a Set in a List

Can implement a set with a list - $O(n)$ space

Union

Use generic merge where

aIsLess = add a into S

bIsLess = add b into S

bothAreEqual = add a into S

Intersection

Use generic merge where

aIsLess = Nothing

bIsLess = Nothing

bothAreEqual = add a into S

Subtraction

aIsLess = ~~Add~~ Nothing a into S

bIsLess = Add b into S

bothAreEqual = Nothing

use in generic merge

Performance

Given the auxiliary methods are $O(1)$ the methods run in linear time

$O(n_A + n_B)$

Set ADT

add(e)

remove(e)

contains(e)

iterator()

union(s)

intersect(s)

subtract(s)

Generic Merge

Requires Auxilary Methods

- aIsLess
- bIsLess
- bothAreEqual

Alg

```
genericMerge(A, B)
  S = empty sequence
  while ¬A.isEmpty() ∧ ¬B.isEmpty()
    a = A.first().element()
    b = B.first().element()
    if a < b
      aIsLess(a, S)
      A.remove(A.first())
    elif b < a
      bIsLess(b, S)
      B.remove(B.first())
    else
      bothAreEqual(a, b, S)
      A.remove(A.first())
      B.remove(B.first())
  while ¬A.isEmpty()
    aIsLess(a, S)
    A.remove(A.first())
  while ¬B.isEmpty()
    bIsLess(b, S)
    B.remove(B.first())
  return S
```

Trees - CS126

What is a tree?

An abstract model of a hierarchal structure

A tree consists of nodes with a parent-child relation.

Tree ADT

Generic methods

size() isEmpty()

iterator() positions()

Accessor methods

root() parent(p)

children(p) numChildren(p)

query methods

isExternal(p) isInternal(p)

isRoot(p)

Tree terminology

Root: node without a parent

Internal node: node with atleast one child

External node: node without children

Ancestors of a node: parent, grandparent, etc

Depth of a node: how many ancestors the node has

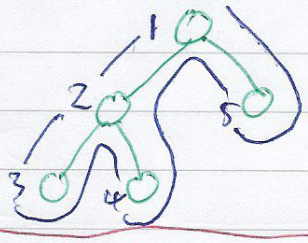
Height of a tree: Max depth of ans node.

Descendants of a node: Child, grandchild, etc.

Subtree: tree consisting of a node and its descendants.

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- preorder:

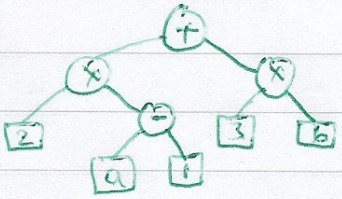


Arithmetic Expression tree

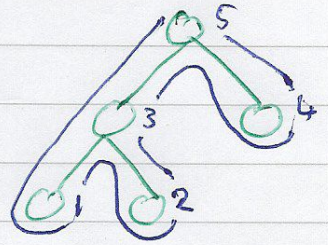
internal nodes - operators

external nodes - operands

e.g. (2x(a-1)+(3x6))



postorder traversal



Binary Trees

A tree with these properties:

- Each internal node has at most 2 children
- The children of a node are an ordered pair (left & right child)

Decision Tree

Internal nodes - questions

External nodes - decisions

Trees 2 - CS126

properties of proper binary trees.

Notation
- n = number of nodes
- e = n° of external nodes
- i = n° of internal nodes
- h = height

properties

$e = i + 1$ $\quad n = 2e - 1$

$h \leq i$ $\quad h \leq (n-1)/2$

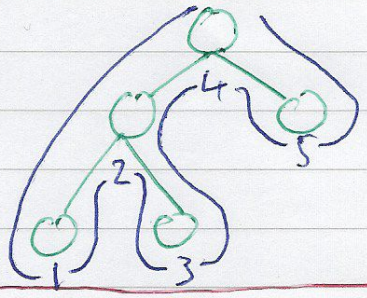
$e \leq 2^h$ $\quad h \geq \log_2 e$

$h \geq \log_2(n+1) - 1$

Binary Tree ADT extends Tree ADT

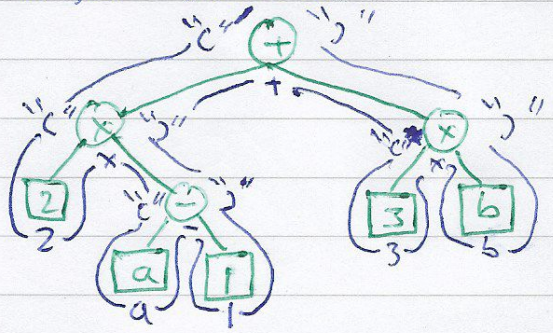
Additional Methods:
- left(p) - right(p)
- sibling(p)

Inorder traversal



Print Arithmetic expressions

Specialisation of inorder traversal.
- Print "(" before traversing left subtree.
- Print ")" after traversing right subtree

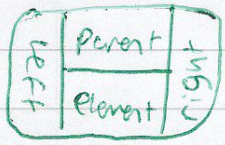


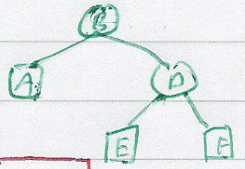
$((2 \times (a-1)) + (3 \times 5))$

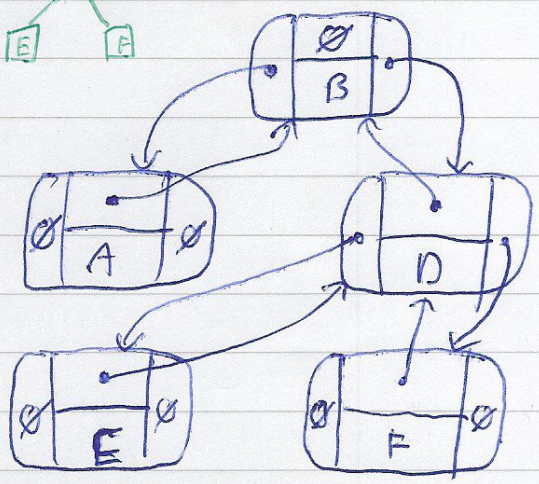
Evaluate Arithmetic Expressions

```
Alg evalExpr(v)
  if isExternal(v)
    return v.element()
  else
    x = evalExpr(left(v))
    y = evalExpr(right(v))
    ◇ = operator stored at v
    return x ◇ y
```

Linked Structure for binary Trees







Array based binary tree

Node p is stored at cell f(p)
- $f(root) = 0$
- if p is left child of q
  $f(p) = 2 \cdot f(q) + 1$
- if p is right child of q
  $f(p) = 2 \cdot f(q) + 2$

Worst case space complexity $O(2^n)$

Priority queues - CS126

What is a priority queue?

- Each element that is stored is assigned a "priority"
- The assigned priority specifies who is leaving first.

Priority Queue ADT

Each entry is a pair

Main methods:

insert(k, v) — min()

removeMin() — size()

isEmpty()

Total Order Relations

Mathematical concept:

- reflexive
- transitive
- antisymmetric
- $a \leq b$ or $b \leq a$ $\forall a, b$

2 distinct entries in the PQ can have the same key.

Entry ADT

key-value pair

Methods:

getKey()

getValue()

Comparator ADT

Encapsulates the action of comparing 2 objects from a given total order

compare(a, b): returns i

- $i < 0$ if $a < b$
- $i = 0$ if $a = b$
- $i > 0$ if $a > b$

Sequence-based priority queue

Implementation with an unsorted list.

Performance:

- insert - $O(1)$
- removeMin - $O(n)$
- min - $O(n)$

Implementation with a sorted list

Performance

- insert - $O(n)$
- removeMin - $O(1)$
- min - $O(1)$

Priority Queue Sorting

1. insert elements 1 by 1 into a PQ.
2. removeMin each element from PQ.

running time is determined by PQ implementation

Selection Sort.

Variation of PQ-Sort where the PQ uses unsorted sequence

runs in $O(n^2)$

Insertion Sort

Variation of PQ-Sort where the PQ uses a sorted sequence

runs in $O(n^2)$

Heaps - CS126

What is a heap?

A binary tree with the following property

Heap-Order:

for every internal node v other than the root, $key(v) \geq key(parent(v))$

Complete Binary Tree:

let h be height of the tree:

- for $i = 0, \ldots, h-1$, there are 2^i nodes at depth i
- at depth h, the external nodes are to the left of the internal nodes.

The last node of a heap is the rightmost node of maximum depth

Heap-Sort

Insert one-by-one into a Heap based PQ.

Extract elements using removeMin.

They are now sorted

Heap-sort runs in $O(n \log n)$

Array-based implementation

element at p is stored in index $f(p)$: If p is root $f(p) = 0$

p left child: $f(p) = 2f(q) + 1$

p right child: $f(p) = 2f(q) + 2$

Space-Complexity $O(n)$

Heaps and PQs

We can use a heap to implement a PQ

We store a key-value pair at each node.

Insertion into a heap

- Insert new node z as the new last node.
- Store k at z
- Restore heap-order using upheap

upheap

Swap k with parent node if $k \leq$ parent.

upheap runs in $O(\log n)$

Removal from a heap

- replace root with last node (w)
- remove w
- restore heap order using down heap

Downheap

- Let p be the root:
  - If no right child, c = left child
  - otherwise, c is minimum child
- If $key(p) \leq key(c)$, heap-order property holds.
- If $key(p) > key(c)$
  - Swap p and c
  - run downheap for subtree rooted where p is now

downheap runs in $O(\log n)$