COMP2123 Data Algorithms and Structures

Lecture 1 Algorithm Analysis

Chapter: [GT 1.1.5-1.1.6, 1.3]

Three abstractions

- Computational problem
  - defines a computational task
    - defines the task you are supposed to solve
    - what input is and what output is needed (but how you get there is not specified)
    - problem statement
  - specifies what the input is and what the output should be
- Algorithm
  - a step-by-step recipe to go from input to output (what your solution does)
  - different from implementation
    - language specific
    - algorithm can be explained in plain English
- Correctness and complexity analysis
  - a formal proof that the algorithm solves the problem (why is what your solution does correct)
    - argue that these are the correct steps to get from the input to the output
  - analytical bound on the resources it uses

Pseudocode

- slightly higher code than python

Control flow

- if ... then ... [else ...]
- while ... do ...
- repeat ... until ...
- for ... do ...
- Indentation replaces braces

Method call

- method (arg [, arg...])

Return value

- return *expression*

Example

- computational code
  - we are given an array A of integers and we need to return the maximum
- algorithm
  - We go through all elements of the array in order and keep track of the largest element found so far (initially $-\infty$). So for each position i, we check if the value stored at $A[i]$ is larger than our current maximum, and if so we update the maximum. After scanning through the array, return the maximum we found.
  - Loop through elements whilst keeping track of largest elements
    - For each position in the array compare to current maximum and update if new element is larger
    - Negative infinity is the mathematical definition of the largest element of an empty set
      - No element will be smaller
- Pseudocode

$max \leftarrow -\infty$
for $i \leftarrow 0$ to $n-1$ do
  if $A[i] > max$ then
    $max \leftarrow A[i]$
return max

- Correctness
  - Invariant
    - Property that your algorithm is maintaining true throughout computation
    - Used to conclude whether algorithm is correct
  - We maintain the following invariant: after the k-th iteration, max stores the maximum of the first k elements.
    - Prove using induction: when k = 0, max is $-\infty$, which is the maximum of the first 0 elements.
  - Assume the invariant holds for the first k iterations, we show that it holds after the (k + 1)-th iteration. In that iteration we compare max to A[k] and update max if A[k] is larger. Hence, afterwards max is the maximum of the first k + 1 elements.
  - The invariant implies that after n iterations, max contains the maximum of the first n elements, i.e., it's the maximum of A.

- Motivation
  - We have information about the daily fluctuation of a stock price
  - We want to evaluate our best possible single-trade outcome
- Input
  - An array with n integer values A[0], A[1], ..., A[n − 1]
- Task

- find indices $0 \leq i \leq j < n$ maximizing

$$A[i] + A[i+1] + \cdots + A[j]$$

Naïve Algorithm

- High level description:
  - Iterate over every pair 0 ≤ i ≤ j < n.
  - For each compute A[i] + A[i+1] + ⋯ + A[j]
  - Return the pair with the maximum value
- Pseudocode

```
curr_val, curr_ans ← 0, (None, None)
for i ← 0 to n − 1 do
  for j ← i to n − 1 do
    {Compute A[i] + A[i + 1] + ··· + A[j]}
    s ← 0
    for k ← i to j do
      s ← s + A[k]
    {Compare to current maximum}
    if s > curr_val then
      curr_val, curr_ans ← s, (i, j)
return curr_ans
```

- More efficient algorithm

We can evaluate $A[i] + A[i+1] + \cdots + A[j]$ faster if we do some pre-processing.

- Pre-compute $B[i] = A[0] + A[1] + \cdots + A[i-1]$ using

$$B[i] = \begin{cases} 0 & \text{if } i = 0 \\ B[i-1] + A[i-1] & \text{if } i > 0 \end{cases}$$

- Iterate over every pair $0 \leq i \leq j < n$.
- For each compute
  $B[j+1] - B[i] = A[i] + A[i+1] + \cdots + A[j]$
- Return the pair with the maximum value

```
curr_val, curr_ans ← 0, (None, None)
B ← new array of size n + 1
B[0] ← 0
for i ← 1 to n + 1 do
  B[i] ← B[i − 1] + A[i − 1]
for i ← 0 to n − 1 do
  for j ← i to n − 1 do
    {Compute A[i] + A[i + 1] + ··· + A[j]}
    s ← B[j + 1] − B[i]
    {Compare to current maximum}
    if s > curr_val then
      curr_val, curr_ans ← s, (i, j)
return curr_ans
```

Efficiency

- An algorithm is efficient if it runs in polynomial time; that is, on an instance of size n, it performs no more than p(n) steps for some polynomial $p(x) = a_d x^d + \cdots + a_1 x + a_0$.
- This gives us some information about the expected behaviour of the algorithm and is useful for making predictions and comparing different algorithms.
    - How long it takes
- Consider worst-case performance
- Running time / number of steps

Asymptomatic growth analysis

- Summarise how T(n) behaves when n increases
- Let T(n) be the worst-case number of steps of our algorithm on an instance of size n.
- Problem
    - figuring out T(n) exactly might be really hard!
    - Also, the fine-grained details are not necessarily that relevant.
- Example
    - $T(n)=4n^2+4n+5$ or $T(n)=5n^2-2n+100$.
        - Which is better?
        - Do the constants matter (recall, one "step" might take a slightly different time based on implementation or architecture details)?
- Insight
    - In both examples, the worst-case number of steps T(n) grew quadratically with n.
    - If n is multiplied by 2, then we expect T(n) to be multiplied by 4.
- More generally: if T(n) is a polynomial of degree d, then doubling the size of the input should roughly increase the running time by a factor of 2d.
- Asymptotic growth analysis gives us a tool for focusing on the terms that make up T (n), which dominate the running time.
- Assumes most steps take generally around the same amount of time

Definition
We say that $T(n) = O(f(n))$ if
there exist $n_0, c > 0$ such that $T(n) \leq cf(n)$ for all $n > n_0$.

$T(n) = 32n^2 + 17n + 32$
$T(n)$ is $O(n^2)$ and $O(n^3)$, but not $O(n)$.

- Running time of algorithm T(n) is big Oh f of n if there exists constants n zero and c both larger than zero such that the running time is upper bounded by f of n multiplied by c once n gets bigger than n zero
- If we have a function, if you multiply that function by a constant than that function times a constant should be higher than the running time of an algorithm if your input gets large enough
- Look at the biggest term – tightest big oh notation running time
    - Big oh is an upper bound on the running time

        - Grows slower than …
- C is any constant
    - N zero can be computed
        - Smallest value that allows c f(n) to be greater then T(n)

Definition
We say that $T(n) = \Omega(f(n))$ if
there exist $n_0, c > 0$ such that $T(n) \geq cf(n)$ for all $n > n_0$.

$T(n) = 32n^2 + 17n + 32$
$T(n)$ is $\Omega(n^2)$ and $\Omega(n)$, but not $\Omega(n^3)$.

- Capital omega
    - Upper bound (grows faster then …)

Definition
We say that $T(n) = \Theta(f(n))$ if
$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

$T(n) = 32n^2 + 17n + 32$
$T(n)$ is $\Theta(n^2)$, but not $\Theta(n)$ or $\Theta(n^3)$.

- Theta
- Upper bound and lower bound match

tl;dr: think of those as
- $T(n) = O(f(n))$: $T(n)$ is "smaller" than $f(n)$ (up to a constant factor)
- $T(n) = \Omega(f(n))$: $T(n)$ is "bigger" than $f(n)$ (up to a constant factor)
- $T(n) = \Theta(f(n))$: $T(n)$ is "equal" to $f(n)$ (up to constants factors)

Asymptotic growth
- Polynomial
    - O(nc), considered efficient since most algorithms have small c
- Logarithmic
    - O(log n), typical for search algorithms like Binary Search
- Exponential
    - O(2n), typical for brute force algorithms exploring all possible combinations of elements
    - Want to avoid

Comparison of running times
- Avoid slower then n^2

size	n	$n \log n$	n^2	n^3	2^n	$n!$
10	< 1s	< 1s	< 1s	<1s	<1s	3s
50	< 1s	< 1s	< 1s	<1s	17m	-
100	< 1s	< 1s	< 1s	1s	35y	-
1,000	< 1s	< 1s	1s	15m	-	-
10,000	< 1s	< 1s	2s	11d	-	-
100,000	< 1s	1s	2h	31y	-	-
1,000,000	1s	10s	4d	-	-	-

Properties of asymptotic growth

- Transitivity
  - If $f = O(g)$ and $g = O(h)$ then $f = O(h)$
  - If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$
  - If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$
- Sums of functions
  - If $f = O(g)$ and $g = O(h)$ then $f + g = O(h)$
  - If $f = \Omega(h)$ then $f + g = \Omega(h)$

Survey of common running times

Let $T(n)$ be the running time of our algorithm.

We say that $T(n)$ is ...	if ...
constant	$T(n) = \Theta(1)$
logarithmic	$T(n) = \Theta(\log n)$
linear	$T(n) = \Theta(n)$
quasi-linear	$T(n) = \Theta(n \log n)$
quadratic	$T(n) = \Theta(n^2)$
cubic	$T(n) = \Theta(n^3)$
exponential	$T(n) = \Theta(c^n)$

What operations take *O*(1) time?

- Constant time:
  - Running time does not depend on the size of the input.
- Assignments (a ← 42)
- Comparisons (=, <, >)
- Boolean operations (and, or, not)
- Basic mathematical operations (+, -, *, /)
- Constant sized combinations of the above (a ← (2 ∗ b + c)/4)

Recall stock trade example

- Constant time in algorithm

```
curr_val, curr_ans ← 0, (None, None)                    O(1)
for i ← 0 to n − 1 do
  for j ← i to n − 1 do
    {Compute A[i] + A[i + 1] + ··· + A[j]}
    s ← 0                                              } O(1)
    for k ← i to j do
      s ← s + A[k]                                     } O(1)
    {Compare to current maximum}
    if s > curr_val then                               } O(1)
      curr_val, curr_ans ← s, (i, j)                   }
return curr_ans                                         O(1)
```

- For loops
  - summations over the number of things that they do
  - example
    - sums from I to j so it takes big oh of (j – i)

```
curr_val, curr_ans ← 0, (None, None)                    O(1)
for i ← 0 to n − 1 do                                                  n−1
  for j ← i to n − 1 do                                                Σ
    {Compute A[i] + A[i + 1] + ··· + A[j]}                             j=i
    s ← 0                                              } O(1)          n−1
    for k ← i to j do                                  } O(j − i)      Σ
      s ← s + A[k]                                     } O(1)          i=0
    {Compare to current maximum}
    if s > curr_val then                               } O(1)
      curr_val, curr_ans ← s, (i, j)                   }
return curr_ans                                         O(1)
```

- running time

$$
\begin{aligned}
T(n) &= O(1) + \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} O(j-i) \\
&= O(1) + \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} O(n) \\
&= O(1) + \sum_{i=0}^{n-1} O(n^2) \\
&= O(1) + O(n^3) \\
&= O(n^3)
\end{aligned}
$$

  - (j – i)'s distance is at most n
    - J can be n and I can be zero
  - Summations
    - At most n terms

    - Summing quadratic gives cubic
- Improved algorithm
    - Cubic to quadratic
    - for loop becomes constant

Lecture 2 Lists

Abstract Data Types (ADTs)

- interface of a data type
    - specifies operations
    - does not specify how they are implemented
- type defined in terms of its data items and associated operations, not its implementation
- interface (people can work with without knowing how things work internally)
- ADTs are supported by many languages, including python
- Example: driving a car
    - Interface/data type
        - Describing how you are supposed to operate the steering wheel and pedals
        - What is supposed to happen you turn the wheel
    - Implementation
        - Motor, electronics
        - Engineer (building car) needs to know about but the user does not
- Benefits
    - Code is easier to understand if different issues are separated into different places.
        - Make sure every function does what it is supposed to do
    - Client can be considered at a higher, more abstract, level.
        - Clients do not need to understand the implementation
    - Many different systems can use the same library, so only code tricky manipulations once, rather than in every client system.
    - There can be choices of implementations with different performance tradeoffs, and the client doesn't need to be rewritten extensively to change which implementation it uses.
        - Change the implementation of the same interface
- Example: Reservation system
    - We have a theatre with 500 named seats, e.g., "N31"
    - What kind of data should be stored?
        - Seats names
        - Seats reserved or available.
        - If reserved, name of the person who reserved the seat.

    - Operations needed? (things are the things I want, user does not need to worry how these things will be implemented)
        - capacity_available() : number of available seats (integer)
        - capacity_sold() : number of seats with reservations
        - customer(x) : name of customer who bought seat x
        - release(x) : make seat x available (ticket returned)
        - reserve(x, y) : customer y buys ticket for seat x
        - add(x) : install new seat whose id is x
        - get_available(): access available seats
- ADT challenges
    - Specify how to deal with the boundary cases
        - what to do if reserve(x, y) is invoked when x is already occupied?
            - Kick out person already there
            - Not let this happen
        - what other cases can you think of?
        - try to include corner cases
    - Do we need a new ADT? Could we use an existing one, perhaps by renaming the operations and tweaking the error-handling?
        - "Adapter" design pattern (see SOFT2201)
        - Could this example be mapped to an ADT you already know?

ADT and Data Structures

- An abstract data type (ADT) is a specification of the desired behaviour from the point of view of the user of the data.
- A data structure is a concrete representation of data, and this is from the point of view of an implementer, not a user.
    - Describe show you will solve the problem from the implementers view
- Distinction is subtle but similar to the difference between a computational problems and an algorithm.

ADT in programming (python)

- ADT is given as an *abstract base class (abc)*
- An abc declares methods (with their names and signatures) usually without providing code and we can't construct instances
    - Inputs, outputs, functions
    - No implementation code
- A data structure implementation is a class that inherits from the abc, provides code for all the required methods (and perhaps others) and has a constructor
- Client code can have variables that are instances of the data structure class and can call methods on these variables

Index-based Lists (List ADT)

- An index-based list (usually) supports the following operations:
    - size() (int) number of elements in the store
    - isEmpty() store (boolean) whether or not the store is empty
    - get(i) return element at index i

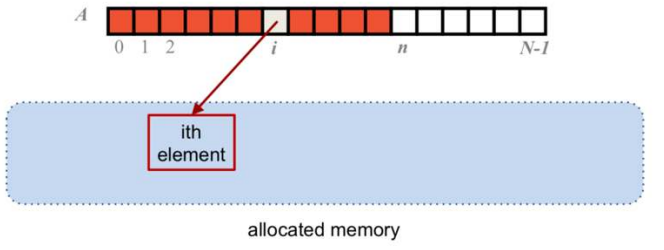
- set(i, e) replace element at index i with element e, and return element that was replaced
- add(i, e) insert element e at index i existing elements with index 3 i are shifted up
- remove(i) remove and return the element at index i existing elements with index 3 i are shifted down

A sequence of List operations:

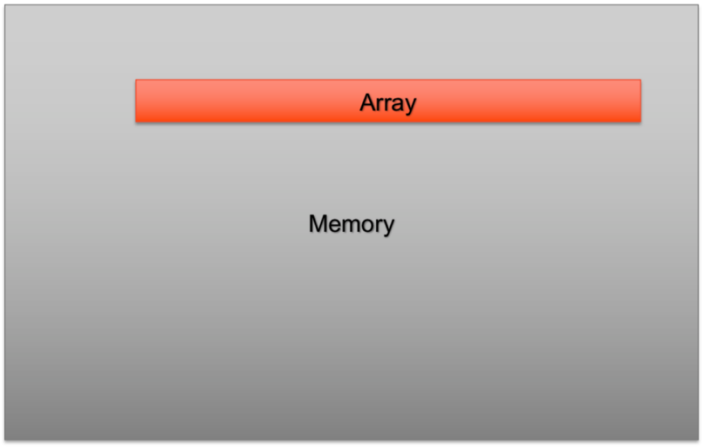
Method	Returned value	List content
add(0,A)	-	[A]
add(0,B)	-	[B, A]
get(1)	A	[B, A]
set(2,C)	"error"	[B, A]
add(2,C)	-	[B, A, C]
add(4,D)	"error"	[B, A, C]
remove(1)	A	[B, C]
add(1,D)	-	[B, D, C]
add(1,E)	-	[B, E, D, C]
get(4)	"error"	[B, E, D, C]
add(4,F)	-	[B, E, D, C, F]
set(2,G)	D	[B, E, G, C, F]

Array based lists

- An option for implementing the list ADT is to use an array A, where A[i] stores (a reference to) the element with index i.
- If array has size N then we can represent lists of size n ≤ N
- Static array
  - Fixed N



- Storage



- Get(i)
  - The get(i) and set(i, e) methods are easy to implement by accessing A[i]
  - Must check that i is a legitimate index (0 ≤ i < n)
  - Both operations can be carried out in in constant time (a.k.a. O(1) time), independent of the size of the array
- Set(I, e)
  - The get(i) and set(i, e) methods are easy to implement by accessing A[i]
  - Must check that i is a legitimate index (0 ≤ i < n)

Pseudo-code for get

```
def get(i):
   # input: index i
   # output: ith element in list
   if i < 0 or i ≥ n then
      return “index out of bound”
   else
      return A[i]
```

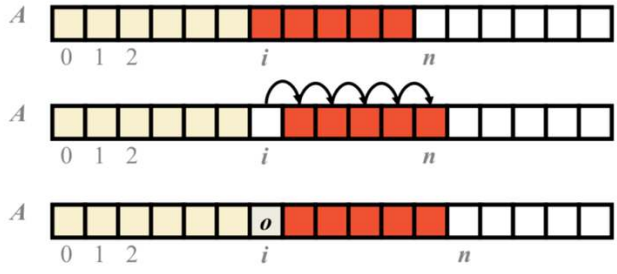
  - Time complexity of this operation is O(1) time, independent of the size of the array (N) or the represented list (n)

Pseudo-code for set

```
def set(i, e):
   # input: index i and value e
   # do: update ith element in list to e
   if i < 0 or i ≥ n then
      return “index out of bound”
   result ← A[i]
   A[i] ← e
   return result
```

  - Time complexity of this operation is O(1) time, independent of the size of the array (N) or the represented list (n)

- Add(I, e)
    - In an operation add(i, e), we must make room for the new element by shifting forward n - i elements A[i], ..., A[n - 1]
    - Must check that there is space (n < N)
    - What is the most time consuming scenario?

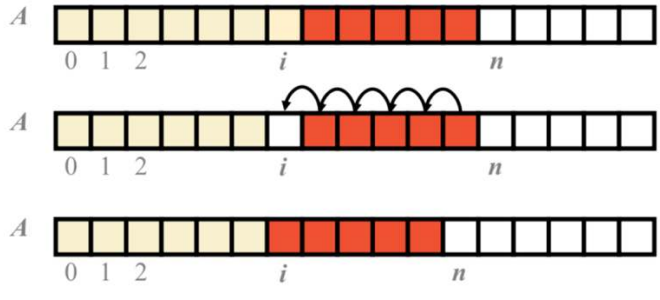


Pseudo-code for insertion

```
def add(i, e):
   if n = N then
      return "array is full"
   if i < n then
      for j in [n-1, n-2, … , i] do
         A[j + 1] ← A[j]
   A[i] ← e
   n ← n + 1
```

    - Time complexity is O(n) in the worst case

- Remove(i)
    - In an operation remove(i), we need to fill the hole left at position i by shifting backward n - i - 1 elements A[i + 1], ..., A[n - 1]
    - Must check that i is a legitimate index (0 ≤ i < n)



Pseudo-code for removal

```
def remove(i):
    if i < 0 or i ≥ n
        return “index out of bound”
    e ← A[i]
    if i < n-1
        for j in [i, i+1, … , n-2] do
            A[j] ← A[j+1]
    n ← n - 1
    return e
```

- Time complexity is O(n) in the worst case

Summary of (static) array-based lists

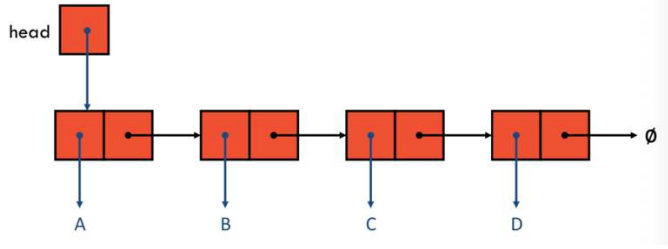
- Limitations:
  - can represent lists up to the capacity of the array (n vs N)
  - cannot represents lists larger than N
- Space complexity:
  - space used is *O*(N), whereas we would like it to be O(n)
- Time complexity:
  - both get and set take *O*(1) time
  - both add and remove take *O*(n) time in the worst case

Positional lists

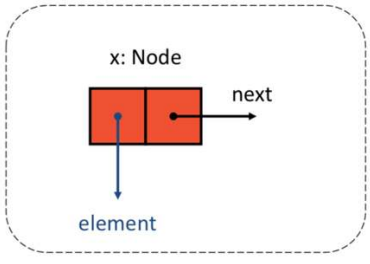
- ADT for a list where we store elements at “positions”
- Position models the abstract notion of place where a single object is stored within a container data structure.
- Unlike index, this keeps referring to the same entry even after insertion/deletion happens elsewhere in the collection.
- Position offers just one method:
  - element() : return the element stored at the position instance
- Operations:

size()	(int) number of elements in the store
isEmpty()	(boolean) whether or not the store is empty
first()	return position of first element (null if empty)
last()	return position of last element (null if empty)
before(p)	return position immediately before p (null if p is first)
after(p)	return position immediately after p (null if p last)
insertBefore(p, e)	insert e in front of the element at position p
insertAfter(p, e)	insert e following the element at position p
remove(p)	remove and return the element at position p

- Singly linked list
  - A concrete data structure
  - A sequence of nodes, each with a reference to the next node
  - List captured by reference (head) to the first node

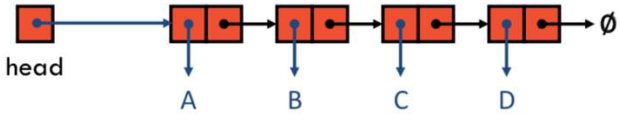


  - Node implements position
    - Each node in a singly linked list stores
      - And its elements and
      - A link to the next node



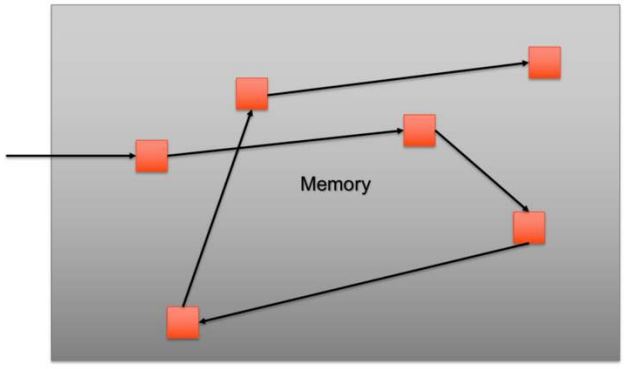
Advice on working with linked structures

- Draw the diagram showing the state.
- Show a location where you place carefully each of the instance variables (including references to nodes).
- Be careful to step through dotted accesses e.g. p.next.next
- Be careful about assignments to fields e.g. p.next = q or p.next.next = r



Linked Lists

- Storage

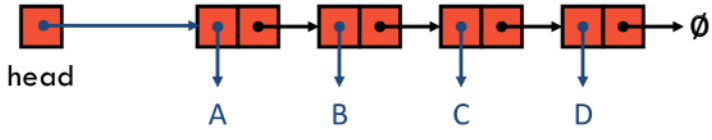


  - Element list stored in a position, with a pointer towards the next element
  - Not stored continuously in memory
  - Can't efficiently access arbitrary
  - Linked lists typically don't support get(), set() functions etc
  - Don't have to move things back and forth to make things continuous like an array
    - Don't care about position of nodes
    - Inserting and removing easier

First()

first() : return position of first element (null if empty)

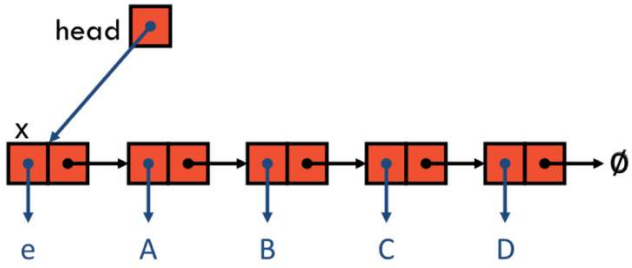
return head



  - Constant time O(1)

insertFirst(e)

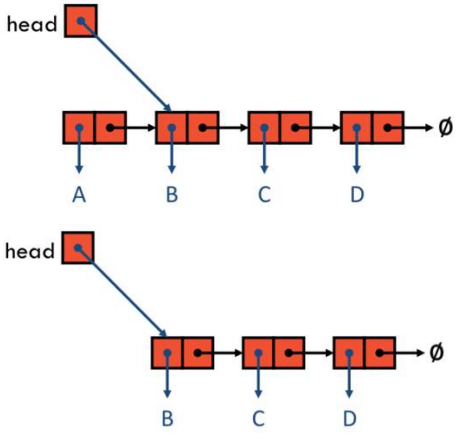
1. Instantiate a new node ***x***
2. Set e as element of x
3. Set x.next to point to (old) head
   - Pointing to node that should be following it
4. Update list's head to point to x



- Constant time O(1)
    - Assigning things

RemoveFirst()

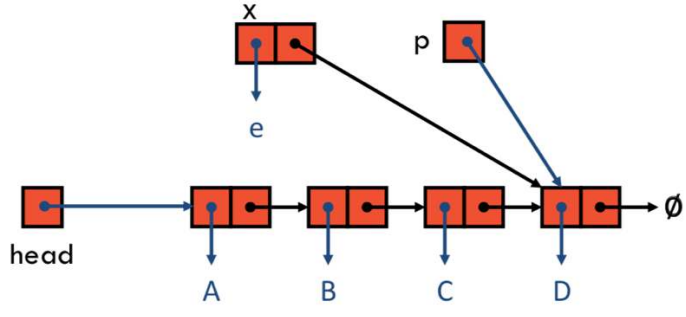
1. Update head to point to next node
2. Delete the former first node



- Constant time O(1)
    - Only doing a constant number of things, nothing depends on the length of the list

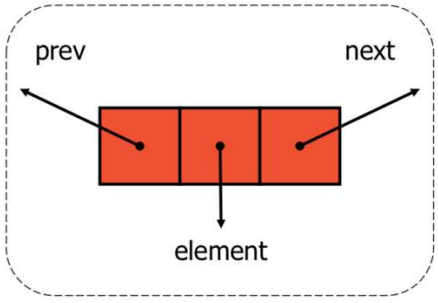
insertBefore(p,e)

- insertBefore(p,e) : insert e in front of the element at position p
    - user must provide position p



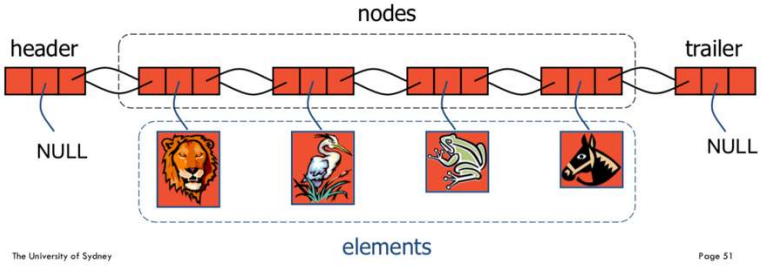
- Next step
    - Find the predecessor of x
        - Follow the links from the “head”
            - Iterate through the list

- Time complexity is O(n)
    - No constant-time way to find the predecessor of a node in a singly linked list
- More efficient way
    - A very natural way to implement a positional list is with a doubly- linked list, so that it is easy/quick to find the position before.
    - Each Node in a Doubly Linked List stores
        - its element, and
        - a link to the previous and next nodes.

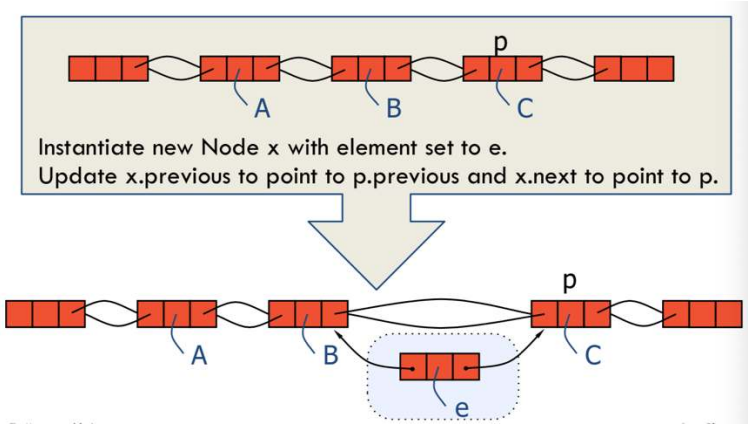


Doubly Linked Lists

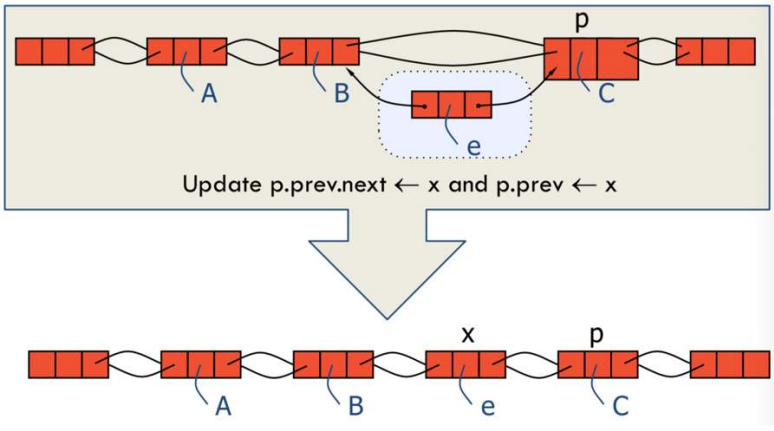
- A concrete data structure
    - A sequence of Nodes, each with reference to previous and to next
    - List captured by references to its Sentinel Nodes
        - Header and trailer (start and end)
        - Don't store any data/element
        - Distinguish between start and end of list and the actual content of the list



InsertBefore(p, e) – step 2



- Predecessor of node x pointing to the previous predecessor of p



- updating pointers to include new node

Pseudocode

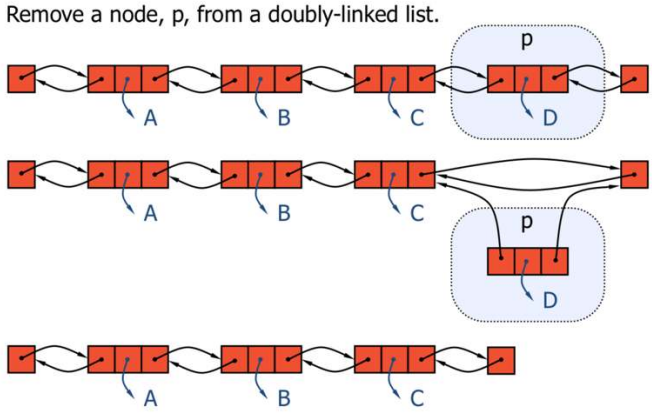
```
def insert_before(pos, elem):
    // insert elem before pos
    // assuming it is a legal pos

    new_node ← create a new node
    new_node.element ← elem
    new_node.prev ← pos.prev
    new_node.next ← pos
    pos.prev.next ← new_node
    pos.prev ← new_node

    return new_node
```

Remove(p)

- Remove a node, p, from a doubly-linked list.



Pseudo-code

```
def remove(pos):
    // remove pos from the list
    // assuming it is a legal pos

    pos.prev.next ← pos.next
    pos.next.prev ← pos.prev

    return pos.element
```

Performance

- A (doubly) linked list can perform all of the accessor and update operations for a positional list in constant time.
- Space complexity is O(n)
    - Constant amount of space per element
- Time complexity is O(1) for all operations

Method	Time
first()	O(1)
last()	O(1)
before(p)	O(1)
after(p)	O(1)
insert_before(p, e)	O(1)
insert_after(p, e)	O(1)
remove(p)	O(1)
size()	O(1)
is_empty()	O(1)

Linked List Implementation

- good match to positional ADT
- efficient insertion and deletion
- simpler behaviour as collection grows

    - do not need to shift things
- modifications can be made as collection iterated over
    - typically not allowed for arrays
- space not wasted by list not having maximum capacity

Arrays Implementation

- good match to index-based ADT
    - specific elements and indexes of list
    - asking for the element at a third of your list
- caching makes traversal fast in practice
    - cycling through array quicker then linked list
- no extra memory needed to store pointers
- allow random access (retrieve element by index)

Iterators

- An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse (travel through) through all the values.
- Abstracts the process of stepping through a collection of elements one at a time by extending the concept of position
- Implemented by maintaining a cursor to the “current” element
    - Allows you to get whatever the next thing is in your iterator
- Two notions of iterator:
    - snapshot freezes the contents of the data structure (unpredictable behaviour if we modify the collection)
        - can’t edit contents
        - contents remain the same
        - static array for example
    - dynamic follows changes to the data structure (behaviour changes predictably)
        - allows you to change data structure as you go through
        - linked lists
        - specifies what changes / its behaviour
- in python
    - iter(obj) returns an iterator of the object collection
    - to make a class iterable define the method __iter__(self)
    - the method __iter__() returns an object having next() methiod
    - calling Next() returns the next object and advances the cursor or raises stopIteraction()

Iterators in Python

```
for x in obj:
   // process x
```

Is equivalent to:

```
it = x.__iter__()
try:
   while True:
      x = it.next()
      // process x
except StopIteration:
   pass
```

Stacks and Queues

- These ADTs are restricted forms of List, where insertions and removals happen only in particular locations:
    - ds follow last-in-first-out (LIFO)
        - allow you to insert and remove things from the end of the list
    - queues follows first-in-first-out (FIFO)
        - allow you to insert things at the end and remove things from the start of the list
- restricted access to list
- why use these
    - simpler/more efficient implementation than lists

Stack ADT

- Main stack operations:
    - push(e): inserts an element, e
    - pop(): removes and returns the last inserted element
- Auxiliary stack 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:

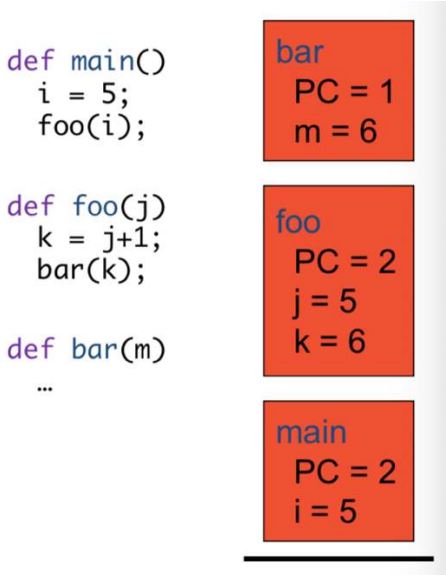
operation	returns	stack
push(5)	-	[5]
push(3)	-	[5, 3]
size()	2	[5, 3]
pop()	3	[5]
isEmpty()	False	[5]
pop()	5	[]
isEmpty()	True	[]
push(7)	-	[7]
push(9)	-	[7, 9]
top()	9	[7, 9]
push(4)	-	[7, 9, 4]
pop()	4	[7, 9]

Stack Applications

- Direct applications
    - Keep track of a history that allows undoing such as Web browser history or undo sequence in a text editor
        - New operation added to stack, if undo is pressed the last thing is removed from the stack
    - Chain of method calls in a language supporting recursion
    - Context-free grammars
- Indirect applications
    - Auxiliary data structure for algorithms
    - Component of other data structures

Method Stacks

- The runtime environment keeps track of the chain of active methods with a stack, thus allowing recursion
- When a method is called, the system pushes on the stack a frame containing
    - Local variables and return value
        - What variables you are using in the current part of code
    - Program counter (PC)
        - Line number that you need to continue executing your program from
- When a method ends, we pop its frame and pass control to the method on top



Parentheses Matches

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “]”
  - correct: ( )(( )){([[ ]])}
  - correct: ((( )(( )){([[ ]])}))
  - incorrect:)(( )){([[ ]])}
  - incorrect: ({[ ])}
  - incorrect: (
- Scan input string from left to right:
  - If we see an opening character, push it to a stack
  - If we see a closing character, pop character on stack and check that they match

Stack implementation based on arrays

- A simple way of implementing the Stack ADT uses an array:
  - Array has capacity N
  - Add elements from left to right
  - A variable t keeps track of the index of the top element
    - Top index of current stack

```
def size()
  return t + 1

def pop()
  if isEmpty() then
    return null
  else
    t ← t – 1
    return S[t + 1]
```

  - Pop operation
    - Define what to do if stack is empty

- Redefine top element (last element) as the previous element, and return that now removed element

- The array storing the stack elements may become full
- A push operation will then either grow the array or signal a “stack overflow” error.

```
def push(e)
  if t = N – 1 then
    return “stack overflow”
  else
    t ← t + 1
    S[t] ← e
```

- Store element at position t

- Performance
    - Space is O(N)
    - Each operation runs in time O(1)
        - Does not depend on the length of array/number of elements
    - Qualifications
        - Trying to push a new element into a full stack causes an implementation-specific exception or
        - Pushing an item on a full stack causes the underlying array to double in size, which implies each operation runs in *O*(1) amortized time (still O(n) worst-case).
            - Creating new array double the size
            - Must go through everything of the existing array to add to the new one
            - Linear time

Queue ADT

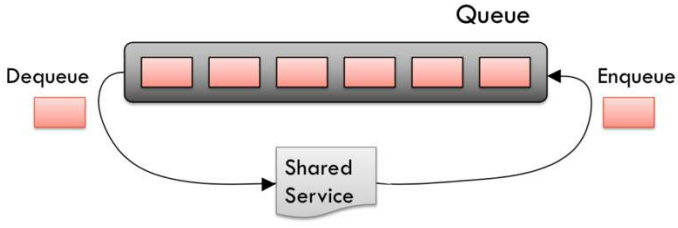
- Main queue operations:
    - enqueue(e): inserts an element, e, at the end of the queue
    - dequeue(): removes and returns element at the front of the queue
- Auxiliary queue operations:
    - first(): returns the element at the front without removing it
    - size(): returns the number of elements stored
    - isEmpty(): indicates whether no elements are stored
- Boundary cases:
    - Attempting the execution of dequeue or first on an empty queue signals an error or returns null

Queue Example

Operation	Output	Queue
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)

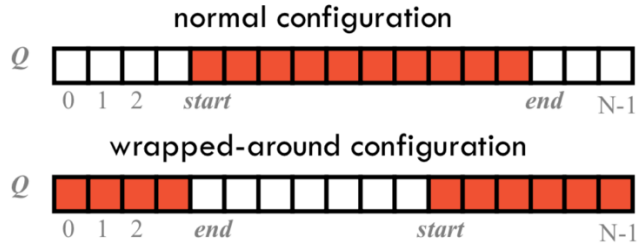
Queue Applications

- Buffering packets in streams, e.g., video or audio
- Direct applications
    - Waiting lists, bureaucracy
    - Access to shared resources (e.g., printer)
    - Multiprogramming
- Indirect applications
    - Auxiliary data structure for algorithms
    - Component of other data structures
- Round robin schedules
    - Implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
        1. e ¬ Q.dequeue()
        2. Service element e
        3. Q.enqueue(e)



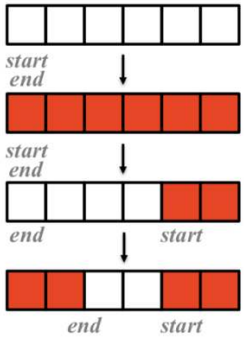
Queue implementation based on arrays

- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
    - start : index of the front element
    - end : index past the last element
    - size : number of stored elements
- These are related as follows end = (start + size) mod N, so we only need two, start and size



How to get in a wrapped-around configuration

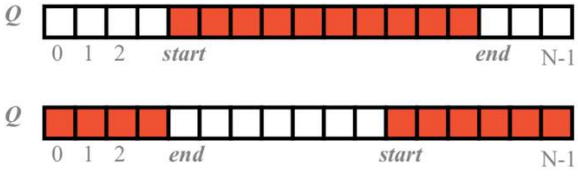
- Enqueue N elements
- Dequeue k < N elements
- Enqueue k' < k elements



Queue Operations: enqueue

- Return an error if the array is full.
- Alternatively, we could grow the underlying array as dynamic arrays do

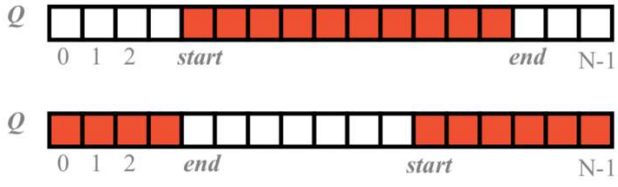
```
def enqueue(e)
  if size = N then
    return "queue full"
  else
    end ← (start + size) mod N
    Q[end] ← e
    size ← size + 1
```



Queue Operations: Dequeue

- Note that operation dequeue returns error if the queue is empty
- One could alternatively return null

```
def dequeue()
  if isEmpty() then
    return "queue empty"
  else
    e ← Q[start]
    start ← (start + 1) mod N
    size ← (size - 1)
    return e
```

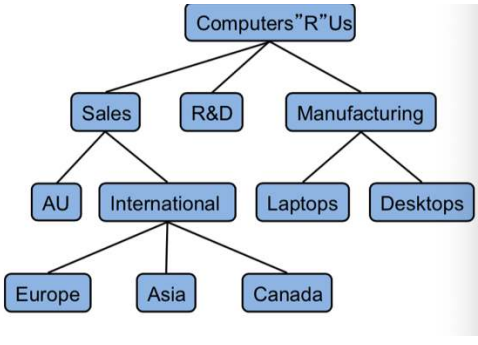


- Performance
  - The space used is O(N)
    - depends on space of static array (size N)
  - Each operation runs in time O(1)

Lecture 3 Trees

What is a tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
  - if u is parent of v, then v is a child of u
  - a node has at most **one** parent in a tree
  - a node can have zero, one or more children
- Root and leaves
- Applications
  - Organization charts
  - File systems
  - Phrase structure

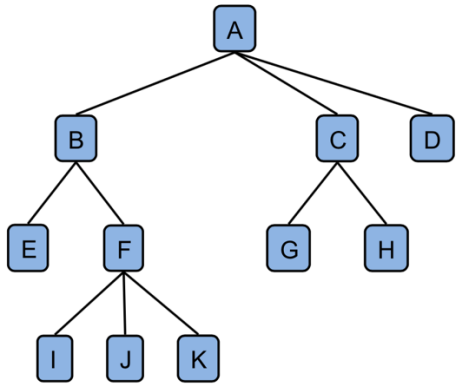


Formal Definition

- A tree T is made up of a set of nodes endowed with parent-child relationship with following properties:
    - If T is non-empty, it has a special node called the root that has no parent
    - Every node v of T other than the root has a unique parent
        - From every node there is only one child-parent relation that you can follow up
    - Following the parent relation always leads to the root (i.e., the parent-child relation does not have “cycles”)

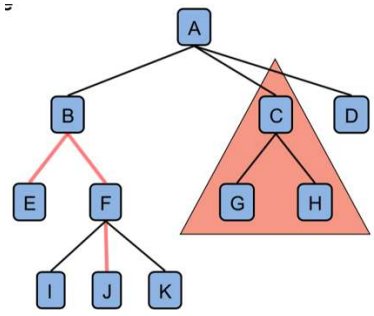
Terminology

- Root: node without parent (e.g. A)
- Internal node: node with at least one child (e.g., A, B, C, F)
- External/leaf node: node without children (e.g., E, I, J, K, G, H, D)



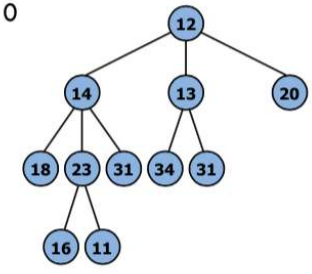
- Ancestors: parent, grandparent, great-grandparents etc (e.g. ancestors of F are A and B)
- Descendants: child, grandchild, great-grandchild etc (e.g. descendant of B are E, F, I, J, K)
- Two nodes with the same parents are sibling

- Depth of node: number of ancestors not including itself (e.g. depth(F) = 2)
- Level: set of nodes with a given depth (e.g. {E, F, G, H} are level 2
- Height of a tree: maximum depth (e.g. 3)
- Substructures of a tree
    - Subtree: tree made up of some node and its ancestors (e.g. subtree rooted at C is {C, G, H})
    - Edge: pair of node (u, v) such that one is the parent of the other (e.g. G and C)
    - Path: sequence of nodes such that 2 consecutive nodes in the sequence have an edge (e.g. <E, B, B, F>)



Examples:

- Node14 has depth … 1
- The tree has height … 3
- Subtree rooted at node 14 has height … 2
- Any subtree of a leaf has height … 0
- The root has depth … 0



Tree Facts

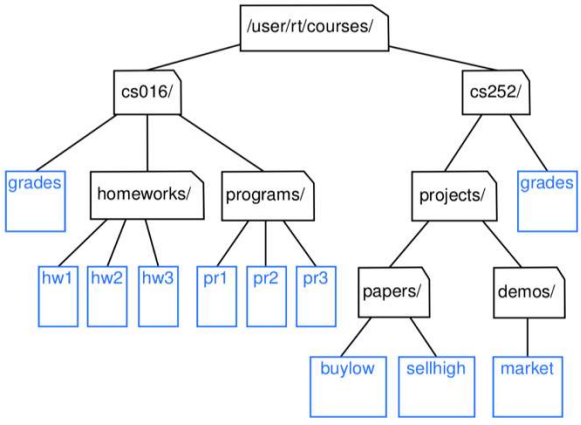
- If node X is an ancestor of node Y, then Y is a descendant of X.
- Ancestor/descendant relations are transitive
- Every node is a descendant of the root
- There may be nodes where neither is an ancestor of the other
  - E.g. F and G
- Every pair of nodes has at least one common ancestor.
  - Common ancestor – ancestor of both nodes
- The lowest common ancestor (LCA) of x and y is a node z such that z is the ancestor of x and y and no descendant of z has that property
  - E.g. for I and Y
    - Common ancestor: nodes X and A
    - LCA : node X

Ordered Tree

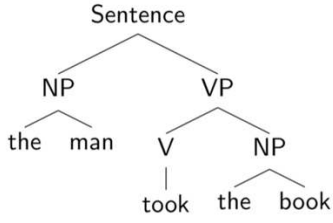
- Sometimes order of sibling's matter
- In an ordered tree there is a prescribed order for each node's children
- In a diagram this ordering is usually represented by the left to right arrangement of the nodes

Application: OS file structure

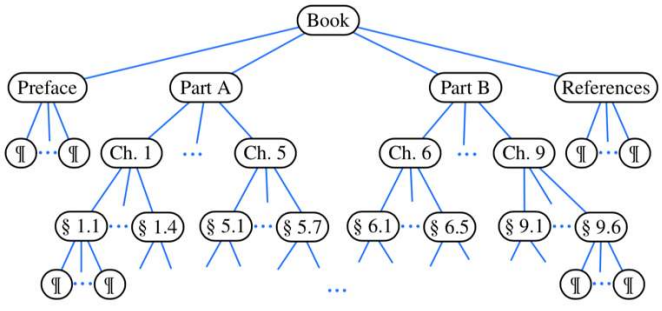
- Ordered children from left to right



Application: Phrase structure tree



Application: Document structure



Tree Abstract Data Type (ADT)

- Position as Node abstraction
    - Similar to linked lists positions
- Supports generic methods:
    - integer size()
    - boolean isEmpty()
    - Iterator iterator()
        - Loop through nodes
    - Iterable positions()
        - Returns the sets of all nodes
        - Then use iterator to iterate through
- Supports access methods: accessing particular nodes
    - Position root()
        - Similar to head of linked list
    - Position parent(p)

        - Figure out the parent of a node
    - Iterable children(p)
        - Figure out the children
        - Returns the set/list of children
            - Can iterate over to go through them individually
    - Integer numChildren(p)
        - Number of children a node has
        - Not descendants just the children of a current node
- Supports query methods:
    - boolean isInternal(p)
        - is the node internal or external
    - boolean isExternal(p)
    - boolean isRoot(p)
- Additional update methods may be defined by data structures implementing the Tree ADT

Node Object

- Node object implementation typically has the following attributes:
    - value: the value associated with this Node
        - name of a department, a number/letter
        - anything you want to store in your node
    - children: set or list of children of this Node
        - pointer to children
    - parent: (optional) the parent of this Node
        - pointer to parents
        - traverse tree in both directions

```
def is_external(v)
   # test if v is a leaf
   return v.children.is_empty()

def is_root(v)
   # test if v is the root
   return v.parent = null
```

- check if a node is the root
    - check if own parent is itself or if parent is null

Traversing trees

- A traversal visits the nodes of a tree in a systematic manner
- When traversing a simpler structure like a list there is one natural traversal strategy (forward or backwards)
- Trees are more complex and admit more than one natural way:
    - pre-order
    - post-order
    - in-order (for binary trees)

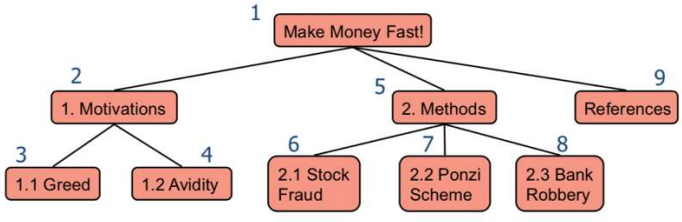
Preorder traversal

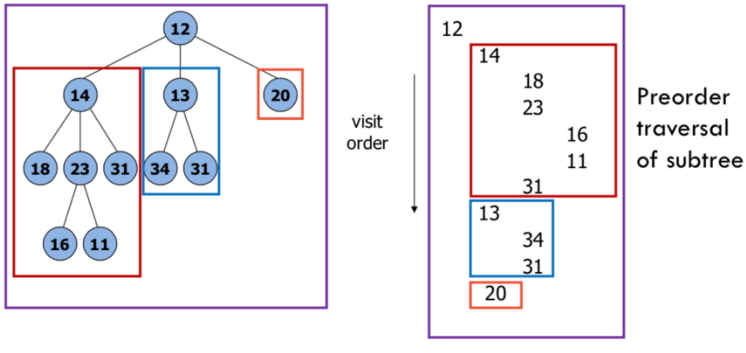
- To do a preorder traversal starting at a given node, we visit the node before visiting its descendants
    - First visit the node you intend to visit, then the children
    - Recursive (calling itself to process the tree)
        - Implies you visit all nodes exactly once
        - Proof you visit all nodes
- If tree is ordered visit the child subtrees in the prescribed order
    - Traverse subtrees in order (left to right)
- Visit does some work on the node: meaning - do this work at visit(v)
    - print node data
    - aggregate node data
    - modify node data

```
def pre_order(v)
   visit(v)
   for each child w of v
      pre_order(w)
```

- Example
    - Nodes are numbered in the order they are visited when we call pre_order() at the root

```
def pre_order(v)
   visit(v)
   for each child w of v
      pre_order(w)
```





Postorder Traversal

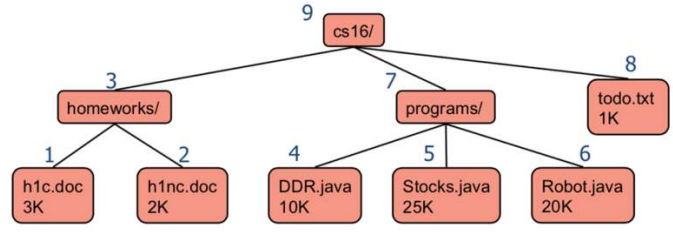
- To do a postorder traversal starting at a given node, we visit the node after its descendants/children
- If tree is ordered visit the child subtrees in the prescribed order
- Visit does some work on the node:

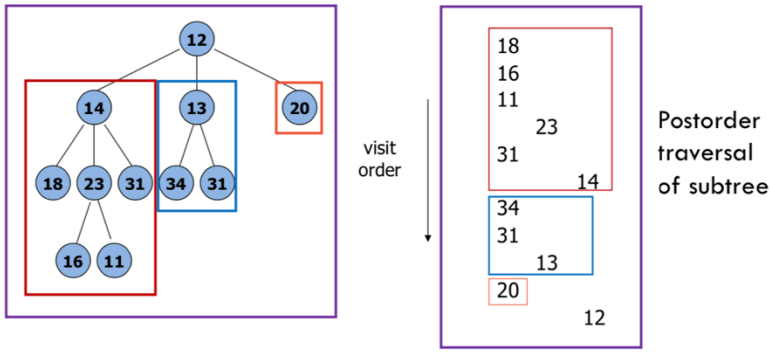
- print node data
- aggregate node data
- modify node data

```
def post_order(v)
   for each child w of v
      post_order(w)
   visit(v)
```

- Example
  - Nodes are numbered in the order they are visited when we call post_order() at the root

```
def post_order(v)
   for each child w of v
      post_order(w)
   visit(v)
```

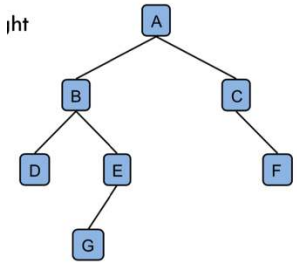




- If you need to pass along info from parent to children use pre-order
  - Root to leaf (top down)
- If you need to compute something where the value of a parent depends on the value of the child use post-order
  - Leaf to root (bottom up)

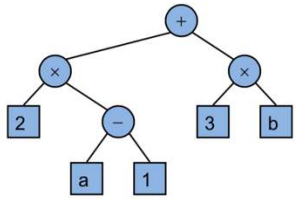
Binary Trees

- A binary tree is an ordered tree with the following properties:
  - Each internal node has at most two children
  - Each child node is labelled as a left child or a right child
  - Child ordering is left followed by right
- The right/left subtree is the subtree root at the right/left child.
- We say the tree is proper if every internal node has two children



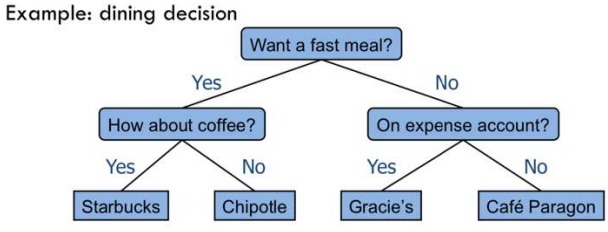
Binary tree application: Arithmetic expression tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
  - Example: Arithmetic expression tree for (2˙ (a - 1) + (3˙ b))



Binary Tree Application: Decision Trees

- Tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- build simple classifiers



Binary Tree Operations

- A binary tree extends the Tree operations, i.e., it inherits all the methods of a tree.
- Additional methods:
  - position leftChild(p)
  - position rightChild(p)
  - position sibling(p)
    - call sibling of LeftChild you get the RightChild
  - return null when there is no left, right, or sibling of p, respectively
- Update methods may be defined by data structures implementing the binary tree

Node Object

- Node object implementation typically has the following attributes:
  - value: the value associated with this Node

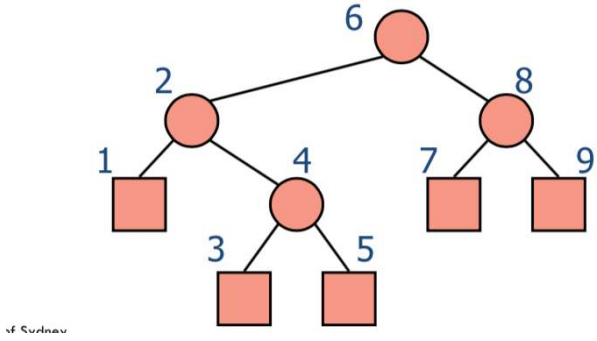
- left: left child of this Node
- right: right child of this Node
- parent: (optional) the parent of this Node

```
def is_external(v)
   # test if v is a leaf
   return v.left = null and v.right = null
```

Inorder Traversal

- To do an inorder traversal starting at a given node, the node is visited after its left subtree but before its right subtree
- Visit does some work on the node:
  - print node data
  - aggregate node data
  - modify node data

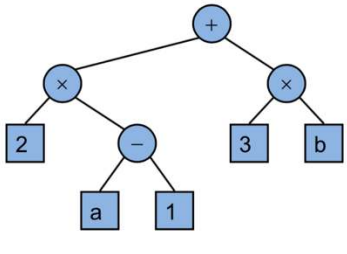
```
def in_order(v)
   if v.left ≠ null then
      in_order(v.left)
   visit(v)
   if v.right ≠ null then
      in_order(v.right)
```



Print Arithmetic Expressions

- Extended inorder traversal:
  - print operand or operator when visiting node
  - print "(" before left subtree – print ")" after right subtree

```
def print_expr(v)
   if v.left ≠ null then
      print("(")
      print_ expr(v.left)
   print(v.element)
   if v.right ≠ null then
      print_expr(v.right)
      print (")")
```

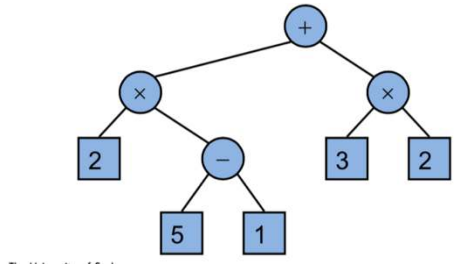


$((2 \times (a - 1)) + (3 \times b))$

Evaluate arithmetic Expressions

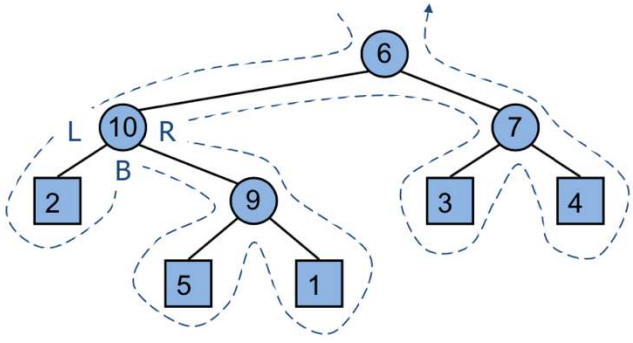
- Extended postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

```
def eval_expr(v)
  if v.is_external() then
    return v.element
  else
    x ← eval_expr(v.left)
    y ← eval_expr(v.right)
    ⊕ ← v.element
    return x ⊕ y
```



Euler Tour Traversal

- Generic traversal of a binary tree. Includes as special cases the preorder, postorder and inorder traversals
- Walk around the tree, keeping the tree on your left, and visit each node three times:
  - on the left (preorder) – from below (inorder)
  - on the right (postorder)
  - visit the leaf nodes three times consecutively

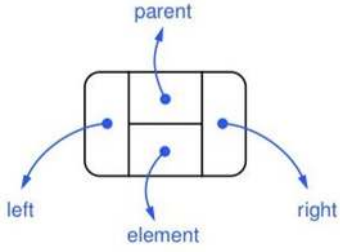


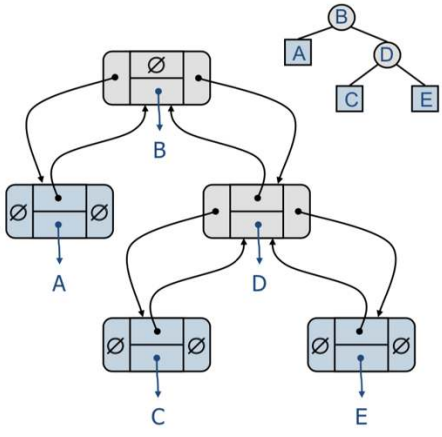
6,10,2,2,2,10,9,5,5,5,9,1,1,1,9,10,6,7,3,3,3,7,4,4,4,7,6

Preorder (first visit): 6, 10, 2, 9, 5, 1, 7, 3, 4
Inorder (second visit): 2, 10, 5, 9, 1, 6, 3, 7, 4
Postorder (third visit): 2, 5, 1, 9, 10, 3, 4, 7, 6

Linked Structure for Binary Trees

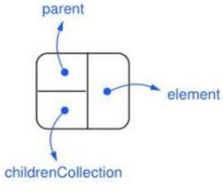
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT

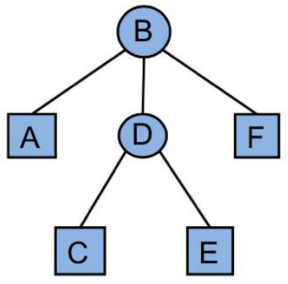


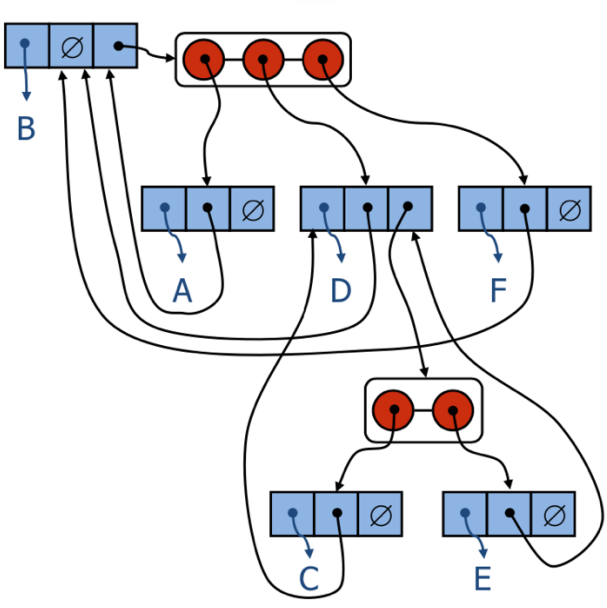


Linked Structure for general trees (implemented a tree)

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children
    - Reference to list of children
      - Doubly linked list for example
- Node objects implement the Position ADT







Examples of recursive code on trees

- recursion is a programming technique using function or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first

Calculating depth

```
def depth(v)
   if v.parent = null then
      return 0
   else
      return depth(v.parent) + 1
```

Calculating height

```
def height(v)
   if v.isExternal() then
      return 0
   else
      h ← 0
      for each child w of v
         h ← max(h, height(w))
      return h + 1
```


Complexity analysis of recursive algorithms on trees

- Sometimes, the method may call itself on all children

    - In worst case, do a call on every node
        - Called on the root, all children means all nodes of tree
    - If the work done, *excluding the recursion*, is constant per call, then the total cost is linear in the number of nodes
        - O(n)
        - Computing the height is a method that will take linear time in the worst case
- Sometimes, the method calls itself on at most one child
    - In worst case, do one call at each level of the tree
        - Follow a single path in the tree
    - If the work done, *excluding the recursion*, is constant per call, then the total cost is linear in the height of the tree
        - Computing the depth is an example

Binary Search Tree

- So far we've been focused on the structure of the tree. The real usefulness of trees hinges on the values we store at each element and how these values are laid out.
- BST is a data structure for storing values that can be sorted. These values are laid out so that an in-order traversal of the BST visits the values in sorted order.
- Can search for elements and insert/delete operations run in O(log n) time provided the tree is "balanced".

Lecture 4: Binary Search Trees

Binary Search Tree

- A binary search tree is a binary tree storing keys (or key-value pairs) satisfying the following BST property
    - For any node v in the tree and any node u in the left subtree of v and any node w in the right subtree of v,
    - key(u) < key(v) < key(w)
- an inorder traversal of a binary search tree visits the keys in increasing order

Implementation