Faculty of Information Technology, Monash University

FIT2004, S2/2016

Week 1: Introduction and Abstract Data Types

Lecturer: Muhammad Aamir Cheema

ACKNOWLEDGMENTS

The slides are based on the material developed by Arun Konagurthu and Lloyd Allison.

Outline

- Part 1: Introduction to the unit
- Part 2: Abstract Data Types (ADTs)

What’s this unit about

- The subject is about problem-solving with computers: algorithms and data structures.
- The subject is not (mainly) about programming.
- The subject just happens to use Python as the programming language in which lab work (etc.) is done. This subject is really language agnostic.
- Algorithms in this courseware will be presented/describe in English, pseudo-code, procedural set of instructions, Python, or even other programming language(s), as convenient.

Why take this unit very seriously?

- The subject is arguably the most important for computer and technology related careers
  - Big companies (e.g., Google, Microsoft, Facebook etc.) actively hunt for people good at algorithms and data structures
  - The things you learn will help you throughout your career
  - Expertise in algorithms and data structures is a must if you want to do research in computer science
- This unit is **CHALLENGING**
  - You have to be on top of it from week 1 – you cannot pass if you think I can cover up the material close to the assessment deadlines
  - Missing lectures, labs or tutorials will require double the efforts to recover

Good News: If you are willing to learn, you will enjoy this unit

FIT2004 Programming Competition: a fun way to learn

Overview of the content

The unit aims to cover

- General problem-solving strategies and techniques, e.g.
  - Useful paradigms, e.g.
    - dynamic programming,
    - divide and conquer, etc.
  - Analysis of algorithms and data structures, e.g.
    - abstract data types,
    - program proof / correctness
    - analysis and estimation of space and time complexity, etc.
- A selection of important computational problems, e.g.
  - sorting,
  - retrieval/searching, etc.
- A selection of important algorithms and data structures,
  - as a tool kit for the working programmer,
  - as example solutions to problems
  - as examples of solved problems, to gain insight into concepts

FIT2004 Staff

Chief Examiner:

Dr. **Arun** Konagurthu, arun.konagurthu@monash.edu
Office: 25 Exhibition Walk, Room 229
webpage: http://www.csse.monash.edu.au/~karun

Lecturer:

Dr. Muhammad **Aamir** Cheema, aamir.cheema@monash.edu
Office: 25 Exhibition Walk, Room 113
webpage: www.aamircheema.com

Tutors:

- **Ammar** Sohail, ammar.sohail@monash.edu
- **Chaluka** Salgado, chaluka.salgado@monash.edu
- **Han** Duy Phan, han.phan@monash.edu
- **Leon** Zhu, liguang.zhu@monash.edu

Course Material

- Your main portal will be, as you already know, the unit's Moodle page: http://moodle.vle.monash.edu/
- Material available on Moodle will include:
  - Introductory Notes
  - Lecture slides
  - Additional references
  - Practical (Lab) sheets
  - Tutorial sheets
- Remember(!) to subscribe to forums and set your e-mail forward

Online reading and research material

- This courseware is built directly on material developed during the previous avatar of FIT2004 (between 1999-2006, when it was called CSE2304). This following link contains a treasure trove of source material on this unit that we will rely on this semester: http://www.csse.monash.edu.au/courseware/cse2304/2006/

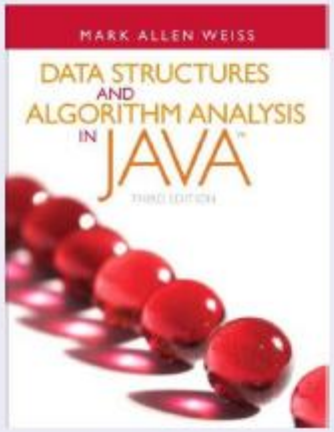
- It is highly recommended that all students navigate through this online material, in addition to the lecture slides on Moodle.

- Following this material, will widen the boundaries of your learning on this topic.

- Especially, there are a number of Javascript-based interactive research material explaining various Algorithms and Data Structures at this link: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS.They are fun and pretty insightful to play around with.

Textbook reference

Mark Allen Weiss, **Data Structures and Algorithm Analysis in Java** 3rd Edition. Addison-Wesley.



This book (and this holds true no matter what the recommend book on this topic is) covers some material not in the lectures and vice versa. Relevant to the course plan this semester are **Chapters 1, 2, 3, 4** (not splay trees), **Chapter 5** (linear, chaining, quadratic), **Chapter 6** (heap and heap sort), **Chapter 7** (relevant sorts), **Chapter 9** (topological sort, paths, spanning Trees, and **Chapter 8** (where relevant), and **Chapter 10**.

(Highly) recommended readings

- This subject has immense practical value for your development into professional programmers, technicians, software engineers, computer scientists, etc. Therefore, you should read beyond what is prescribed for this unit.

- Additional books you might want to refer to (from time-to-time, even beyond this unit):

  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliord Stein. *Introduction to Algorithms*. 3rd Edition. The MIT Press & Mc Graw Hill.
  - Donald Knuth, *The Art of Computer Programming*, Pearson. (Pretty expensive, but good-value-for-money for serious programmers; library has copies!)

Course Structure

- **Lectures**
  - Mondays 17:00 to 19:00 @CL_21Rnf/S7
- **Practicals/Pracs/Labs** (refer to your timetable in allocate+)
  - Every week, starting week 2
  - Pracs in weeks 2, 3, 5, 7, 9, 11 are NOT assessed and conducted over a 2 hour slot (except the 'practice' lab in wk 2 which will go on for the full 3 hrs).
  - Pracs in weeks 4, 6, 8, 10, 12 are ASSESSED and conducted over a 3 = 1.5 + 1.5 hour slot (where the last 1.5 hour is used for marking/assessment).
- **Tutorials/Tutes** (refer to your timetable in allocate+):
  - 1 hour fortnightly, starting week 3.
  - Weeks 3, 5, 7, 9, 11
  - This 1 hour slot will immediately precede non-assessed pracs in weeks 3, 5, 7, 9, 11

Tutorials (in more details)

- 1 hour fortnightly, compulsory
- Weeks 3,5,7,9,11
- Locations and times (see Allocate+)
- Notes are made available during semester and may be downloadable from Moodle
- Tutorials must be attempted during self-study beforehand
- Tutorials provide, among other things, background to pracs, especially those that are assessed.

Pracs (in more details)

- Held weekly starting week 2, all the way through to week 12.
- Locations and times see Allocate+.
- 3 hour Assessed lab in Weeks 4, 6, 8, 10 and 12. The last 1-1.5 hour during these weeks will be used for marking.
- Conducted by your Tutors
- Organize your computer account before the first prac in week 2, if you don't already have one.
- **Important:** All programming questions in your lab have to be done in Python programming language.

Marks and hurdles - IMPORTANT

To pass FIT2004

- Your marks must average to at least 50% of the total marks for this unit, and
- you must pass each of the hurdles

Prac class assessments = 30% of total semester marks

- Total in-semester marks =30 out of 100, with 6 marks per assessed prac in weeks 4,6,8,10,12 .
- HURDLE: You should score at least 40% of the in-semester marks (i.e., 12 out of 30 marks)

End-of-Semester Examination = 70% of total semester marks

- End-of-Semester (written) exam carries 70 marks.
- HURDLE: You should score at least 40% of the exam marks (i.e., 35 out of 70 marks)

Missed Prac

If you miss an assessed prac, you will be marked as ABSENT, unless you do TWO things:

1. Arrange to attend an alternative prac the same week (with the approval of the Lecturer and Tutor), AND
2. Email to your Lecturer **and** Tutor, the following details
   - Name:
   - Student ID number:
   - Date of replacement Prac:
   - Time and Location of the regular prac:
   - Time and Location of the replacement prac:

NOTE: The subject line of your email should read

FIT2004: Replacement lab request for <your student number>

Missed Prac

If you are unable to attend a replacement prac due to an illness or emergency, then

- Obtain supporting documentation (e.g., medical certificate)
- Fill out a Special Consideration Form and hand it in (with supporting documentation) to me (Office 113, CL 25Exhibition; slide it under my door if I am not in).
- Read Monash University's [Special Consideration policy].

Plagiarism and Cheating

- Monash University takes plagiarism and cheating very seriously. There are severe penalties for them.

- Read Monash University's [Plagiarism and cheating Policy]

- In a nutshell, Plagiarism is legitimately using someone else's work, but not acknowledging it. Cheating is pretending that someone else's work is your own, in order to gain an unfair advantage.

- It is OKAY to work together in trying to understand concepts. Peer Assisted learning (PAL) is fun and a great way to cross-fertilize each others' thinking -- but each student must be conscientious in his work, write the entire assignments alone, and be able to explain and modify it on request

Student responsibilities

Responsibilities

- Be regular to the weekly lecture.
- Catch up missed lectures promptly through recordings on MULO.
- Attendance to labs and tutes are compulsory
- Do not leave the assessed lab before your lab demonstrator has marked your assignment.
- Prepare during self-study (nominally 8 hours/week -- more is better).
- This subject is best understood practically. So, assimilating various concepts and practicing them (a lot!) is the key to success here.

Etiquette

- No noise and distractions during the lectures
- Turn your mobile devices to silent mode.

FIT2004: Programming Competition

- Contest Format
  - Individual contest requiring you to write program to solve algorithmic problems
  - Three rounds. Tentative schedule:
    - Round 1: Week 3
    - Round 2: Week 7
    - Round 3: Week 11
  - 3-6 problems in each round; each round will accept submissions for 1-2 weeks
  - Solutions will be automatically marked by an online judging system
  - Your score is the number of problems you solve
  - Contestants will be ranked according to their scores, and ties will be broken based on **penalty** (explained below)
  - Each incorrect submission carries a penalty. But the penalty is only applied if you finally solve the problem (e.g., say you solve problem #1 in 3 attempts and you have 3 failed attempts for problem #2 which you were unable to solve. Your score will be 1 and penalty will be 2 failed attempts on problem #1.)
- Registration: to be open soon (keep an eye on Moodle)
- Prizes: TBA + (Certificates for top-5 contestants)
- Practice: www.codeforces.com, https://uva.onlinejudge.org/

MARS: Monash Audience Response System

1. Visit http://mars.mu on your internet enabled device
2. Log in using your Authcate details (not required if you're already logged in to Monash)
3. Touch the + symbol
4. Enter the code for your unit: Y4R44G
5. Answer questions when they pop up

MARS: Multiple Choice Questions

1 + 1 = ?

A. 2
B. 10
C. Not sure

Hint: There are 10 types of people in this world. Those who understand Binary and those who don't.

MARS: Text-based questions

Where do you live? (write the name of the suburb)

Anonymous surveys

- Throughout the semester, I will send a few anonymous surveys to get your feedback on things that can be improved in this unit

- Please do fill these surveys (will take < 5 mins)

Short break

Part 2: Abstract Data Types

- Thinking about Abstract data types (ADTs) algebraically and formally.
- Algebraic data type: [List]
- Algebraic data type: [(Binary) Tree]
- Reasoning about algorithms on such ADTs formally

Reminder

- Subject is about problem solving...
- ...and not programming (in Python) as such.

What and Why ADTs?

An Abstract Data Type provides a mathematical model to define a data type and its possible values, operations and behavior.

- Definition
- Operations
- Rules
- Functions

Advantages

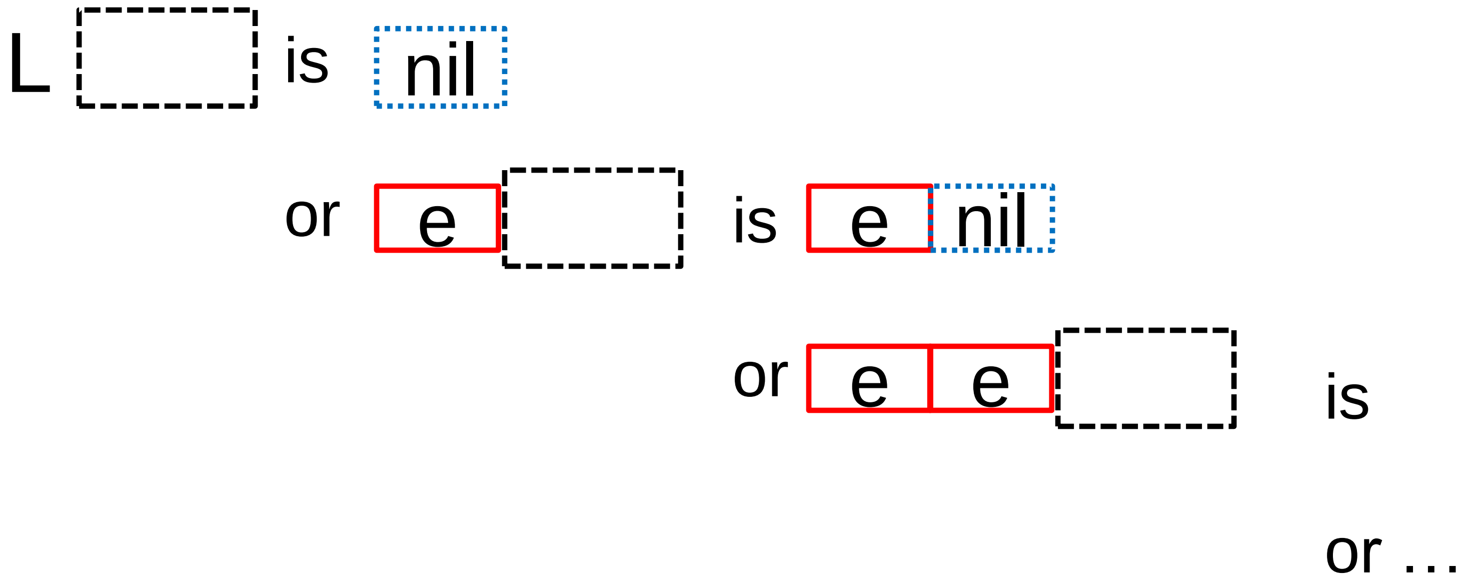
- It is much easier to reason formally about ADTs and abstract algorithms than about structures in algorithms in some specific language
- Abstraction guarantees that any implementation of an ADT guarantees certain **properties** and **abilities**

List ADT: Definition

```
TYPES
        type list e = nil | cons(e * (list e))
```

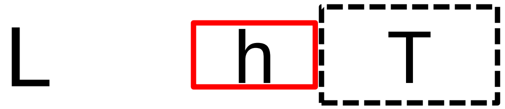
In plain English:

A list of elements of type e is (=) either empty (nil) or (|) is constructed (cons) using an element of type e AND (*) a list.



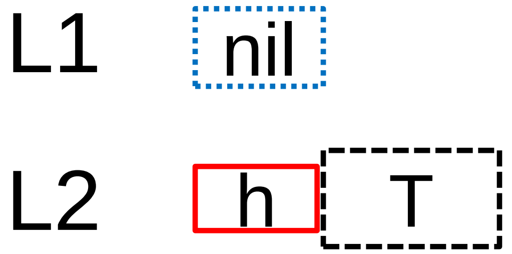
List ADT: Operations

```
Operations
   isNull : list e -> boolean ;OR: isEmpty, null etc.
   head : list e -> e               ;OR: hd, first , etc.
   tail : list e -> list e          ;OR: tl, rest , etc.
   cons : e * (list e) -> list e
```



List ADT: Rules

```
RULES
 isNull(nil) = true
 isNull(cons(h,T)) = false
 head(cons(h,T)) = h
 head(nil) = error
 tail(cons(h,T)) = T
 tail(nil) = error
```



```
EXAMPLES:
isNull(L1) = true
isNull(L2) = false
head(L2) = h
head(L1) = error
tail(L2) = T
tail(L1) = error
```

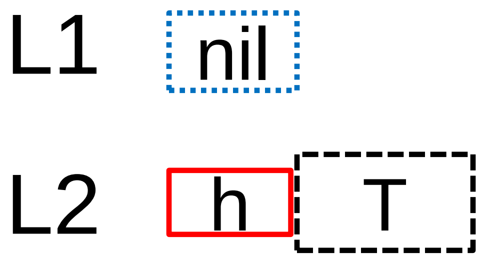
List ADT: Functions

length function

```
        length(nil) = 0
                | length(cons(h,T)) = 1 + length(T)
```

Observations

- the length function has been recursively defined here
- While there are (potentially) infinitely many lists, there are only two cases that the length function must consider.
- Note, most functions on lists have the same general structure as length. If one case is missing the function is probably INCOMPLETE!



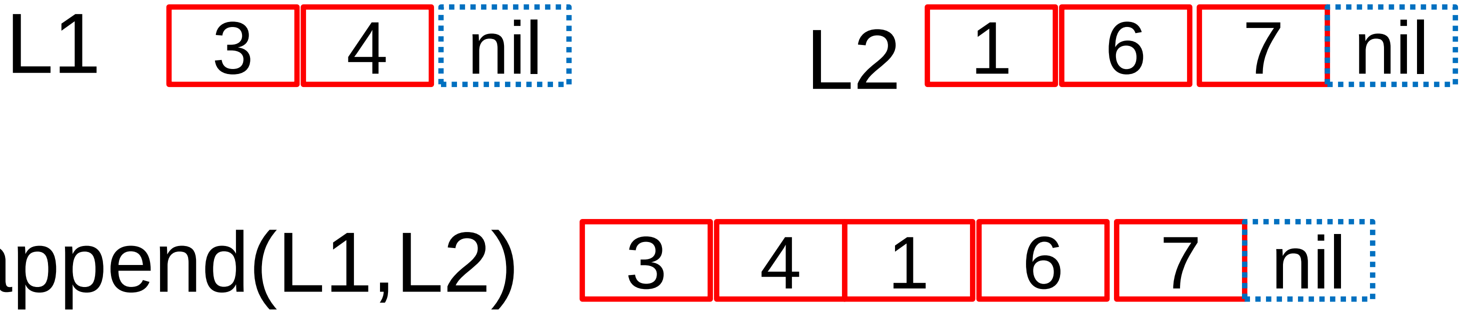
EXAMPLES:

```
length(L1) = 0
length(L2) = 1 + length(T)
```

List ADT: Functions

Append function

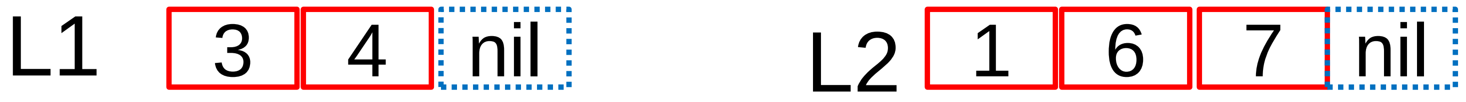
```
      append(nil, L2) = L2
      | append(L1,L2) = append(cons(h1,T1),L2)
                                    = cons(h1,append(T1,L2))
;COMMENT: L1, L2 are two lists. h1 = head(L1). T1 =
tail(L1).
```



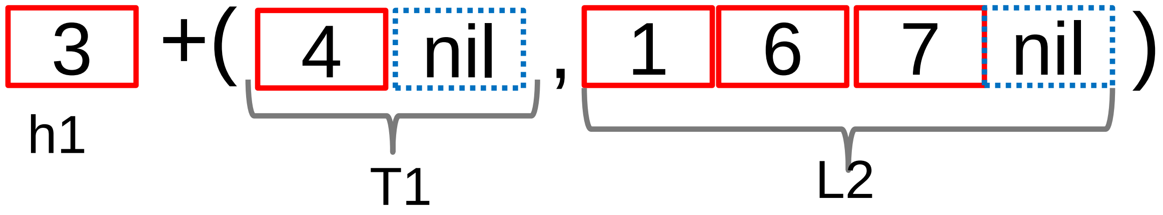
List ADT: Functions

Append function

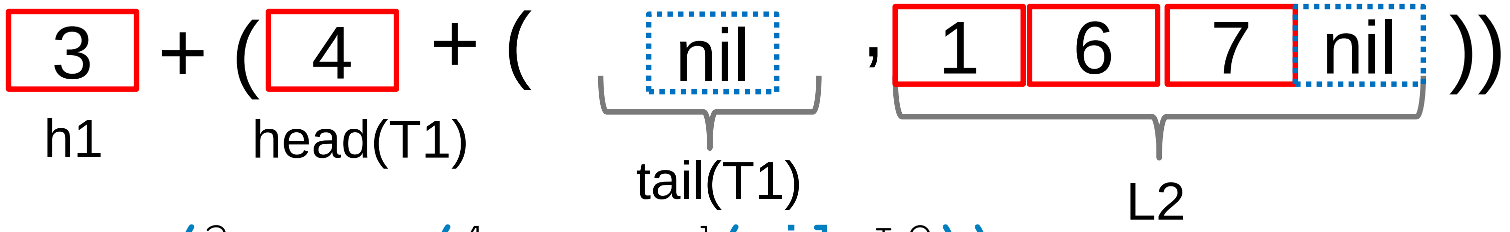
```
        append(nil, L2) = L2
      | append(L1,L2) = append(cons(h1,T1),L2)
                                = cons(h1,append(T1,L2))
```



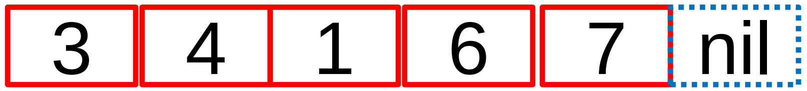
```
cons(3,append(T1,L2))
```



```
cons(3, cons(4,append(tail(T1),L2))
```



```
cons(3, cons(4,append(nil,L2))
```



List ADT: Functions

Append function

```
        append(nil, L2) = L2
      | append(L1,L2) = append(cons(h1,T1),L2)
                                = cons(h1,append(T1,L2))
```

What is the time complexity of the append function?

A. **O(|L1|)**

B. **O(|L2|)**

C. **O(|L1| + |L2|)**

D. **O(|L1| X |L2|)**

E. **None of the above**

Note: |L| is the number of elements in a list

IMPORTANT: Some implementations offer better complexity. ADT defines what a function does not how it is done.

Proofs with ADTs

Theorem

- The append operation is associative:

```
append(L1,(append(L2,L3))) = append(append(L1,L2),L3)
```

Prove it by induction ???

Revision: Proof by induction

2 steps

1. Prove the base case, e.g., for the first state
2. Assume the proof holds for a state k. Show that it also holds for the next state k+1.

Theorem

Let S(N) be the sum of the integers from 1 to N

Prove that S(N) = N(N+1)/2

Step 1: Base Case: N = 1

S(1) = 1 and N(N+1)/2 is 1 for N=1

Step 2: Inductive step

Assume it holds for a positive integer k

S(k) = k(k+1)/2

Show that S(k+1) = (k+1)(k+2)/2

continued on next slide…

Revision: Proof by induction

Theorem

```
Let S(N) be the sum of the integers from 1 to N
Prove that S(N) = N(N+1)/2
```

Step 2: Inductive step

```
Assume it holds for a positive integer k
S(k) = k(k+1)/2
Show that S(k+1) = (k+1)(k+2)/2


S(k+1) = 1 + 2 + 3 + … + k+1
S(k+1) = 1 + 2 + 3 + … + k + k+1
S(k+1) = k(k+1)/2    +   k+1
S(k+1) = (k+1)(k+2)/2
```

For more details, watch it on Khan Academy (click here)

Proof: Append is associative

Theorem

- The append operation is associative:

```
append(L1,(append(L2,L3))) = append(append(L1,L2),L3)
```

Step 1: Base Case (L1 is nil)

```
append(nil ,(append(L2,L3)))
= append(L2,L3)
= append(append(nil,L2),L3)
```

Step 2: Inductive step

Suppose L1 is cons(h1,T1). Assume that the proof holds for the state T1, e.g.,

```
append(T1,(append(L2,L3))) = append(append(T1,L2),L3)
```

Show that it holds for the next state where the next state is when the list L1 is cons(h1,T1)

continued on next slide …

Proof: Append is associative

Theorem

```
append(L1,(append(L2,L3))) = append(append(L1,L2),L3)
```

Step 2: Inductive step

Assume that the proof holds for the state T1, e.g.,

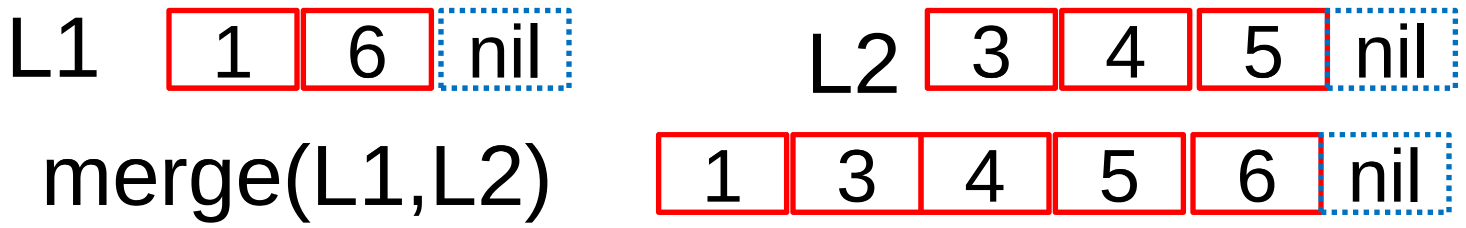
```
append(T1,(append(L2,L3))) = append(append(T1,L2),L3)
```

Show that it holds for the next state where the next state is when the list L1 is cons(h1,T1)

```
append(L1,append(L2,L3))
= append(cons(h1,T1), append(L2,L3))
= cons(h1,append(T1,append(L2,L3)))
;COMMENT: use the assumption
= cons(h1, append(append(T1,L2),L3))
= append(cons(h1,append(T1,L2)), L3)
= append(append(cons(h1,T1),L2), L3)
= append(append(L1,L2), L3)
```

List ADT: Functions

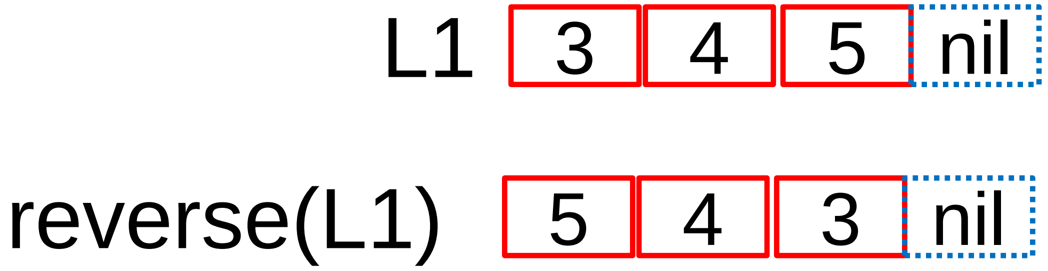
```
merge function;produces a sorted list if L1&L2 are sorted
 ;COMMENT: when at least one list is null
 merge(nil, nil) = nil
 | merge(nil, L1) = L1
 | merge(L1, nil) = L1
 ; COMMENT: when L1 not null and L2 not null
 | merge(L1, L2) =
 if head(L1) < head(L2) then
      cons(head(L1), merge(tail(L1),L2))
 else ;if head(L1) >= head(L2)
      cons(head(L2), merge(L1, tail(L2)))
```



List ADT: Functions

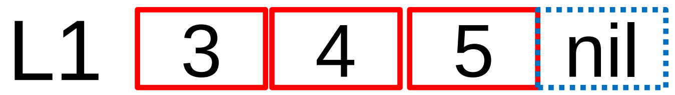
reverse function

```
 reverse(nil) = nil
 | reverse(cons(h,T)) = append(reverse(T),cons(h,nil))
```

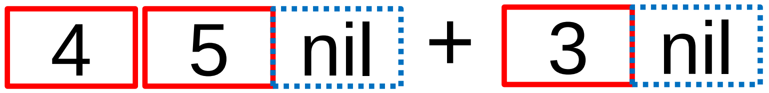


List ADT: Functions

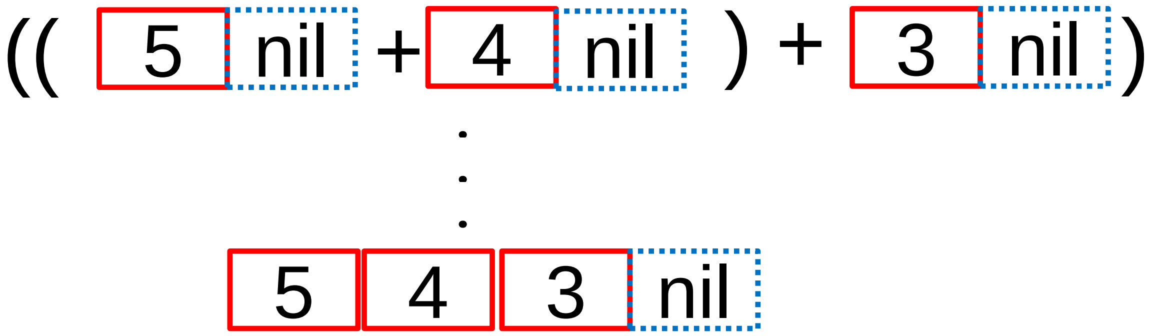
```
reverse function
 reverse(nil) = nil
 | reverse(cons(h,T)) = append(reverse(T),cons(h,nil))
```



```
append(reverse(T1),cons(h1,nil))
```



```
append(append(reverse(tail(T1),cons(head(T1),nil))
,cons(h1,nil))
```



List ADT: Functions

reverse function

```
 reverse(nil) = nil
 | reverse(cons(h,T)) = append(reverse(T),cons(h,nil))
```

What is the time complexity of reverse function?

A. O(N)
B. O(N^2)
C. O(N log N)
D. None of the above

Note: N is the number of elements in the list

Comments on recursion and iterations

- Many of the operations on lists (and other recursive data types) are naturally expressed as recursive routines.
- Recursive routines have fewer state variables than iterative routines and are frequently easier to prove correctness. However recursion does require system-stack space to operate.
- Iterative versions of many routines, particularly for the simpler operations, are straightforward to implement so often preferred, but are much harder to prove correctness.
- We are not going to discuss merits and demerits of recursion now. Suffice it to say that recursion and iteration are both powerful and useful techniques.

Implementing Lists

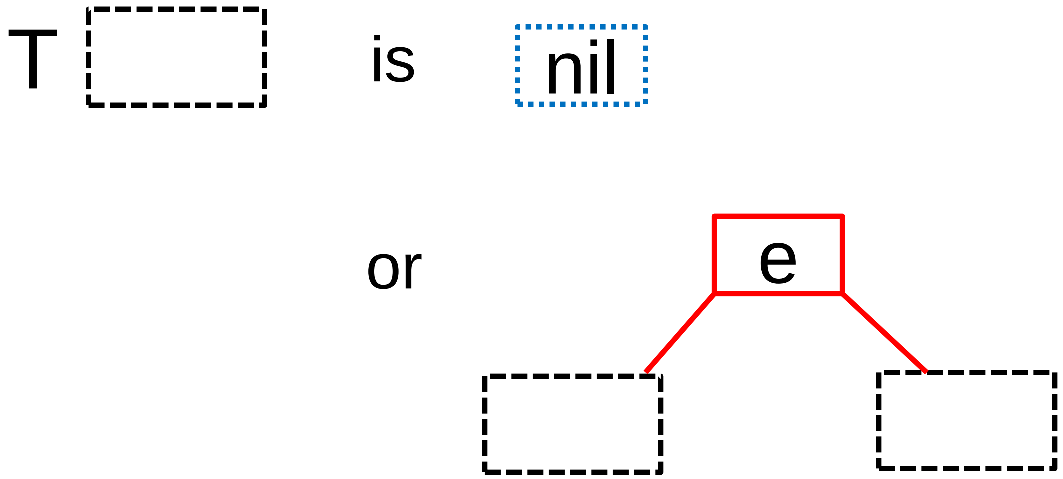
- The most natural way to implement lists is by means of records/structs and pointers/links.
- They can also be implemented using arrays.
- You (might) have done this exercise in FIT1008.
- In any case, we will implement this as an exercise, to warm up your programming abilities.

The Abstract Data Type: Trees

- Trees are natural structures for representing certain kinds of hierarchical data.
- A (rooted) tree consists of a set of nodes (or vertices) and a set of arcs (or edges).
- Each arc links a parent node to one of its children.
- A special root node has no parent.
- Every other node has exactly one parent.
- It is possible to reach any node by following a unique path of arcs from the root.
- If arcs are considered bidirectional, there is a unique path between any two nodes.
- The simplest kind of tree is a binary tree where each parent has at most two children.

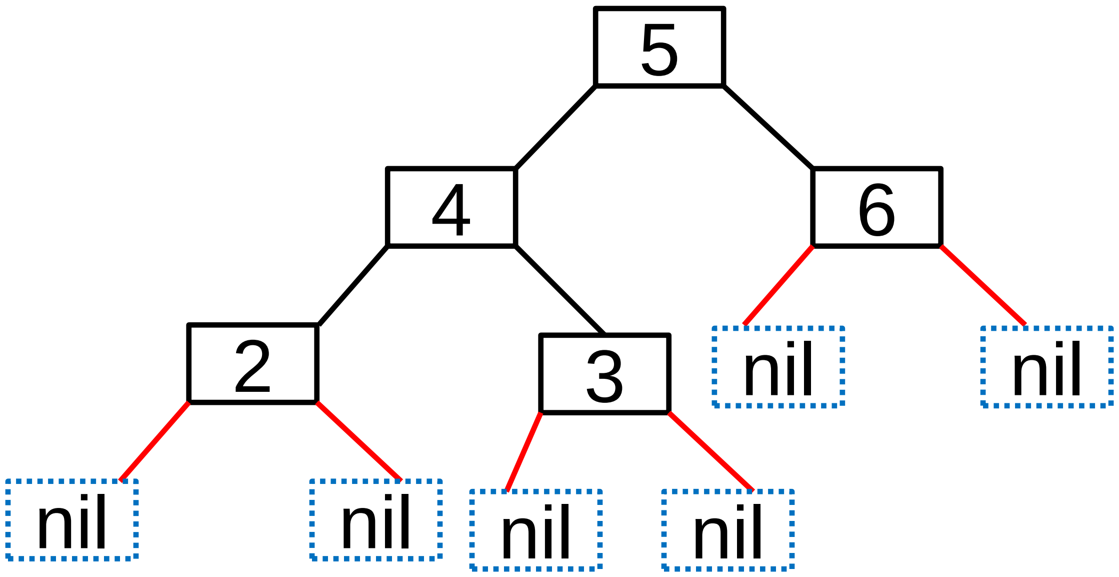
ADT Trees: Definition

```
TYPES:
    type tree e = nilTree | fork e * (tree e) * (tree e)
```



ADT Trees: Operations

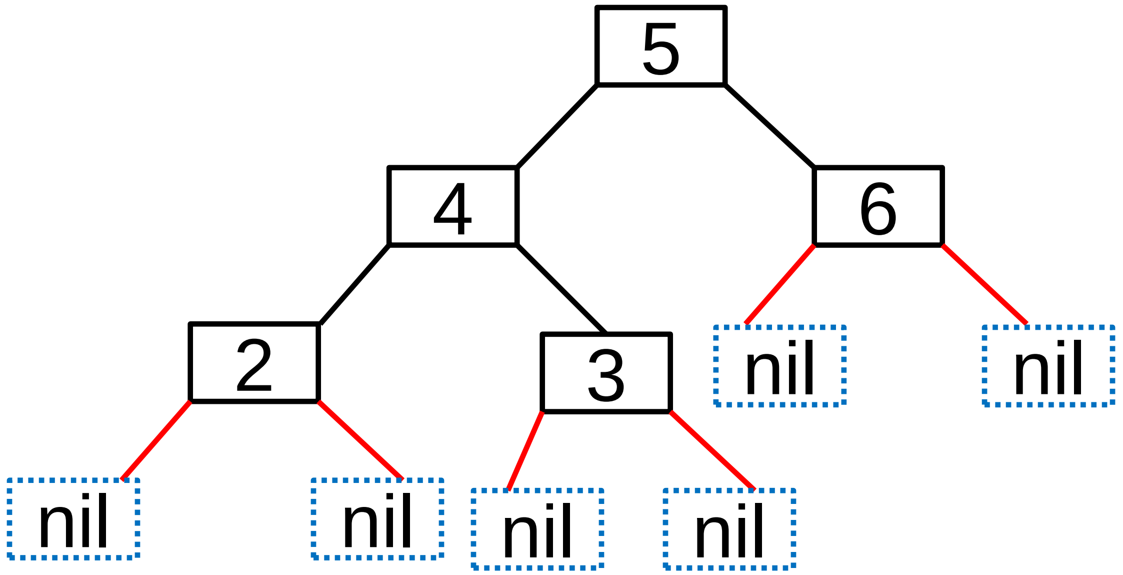
```
empty: tree e -> boolean
leaf : tree e -> boolean
fork : e * tree e * tree e -> tree e
left : tree e -> tree e
right: tree e -> tree e
contents: tree e -> e
```



ADT Trees: Functions

height function

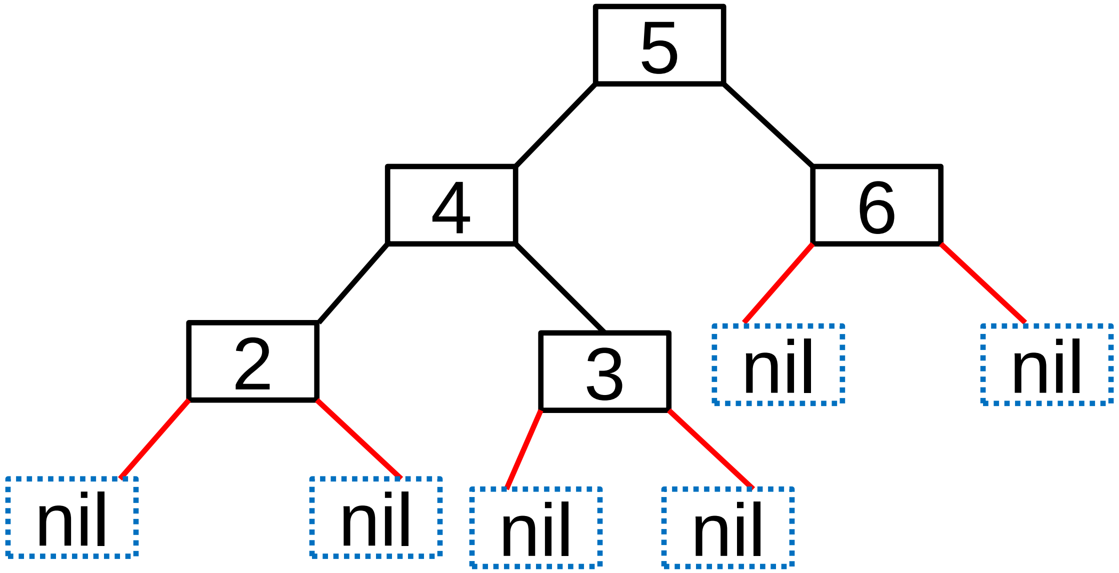
```
height(nilTree) = 0
| height(fork(e,TL,TR)) = 1+max(height(TL), height(TR))
```



What is the height of the above tree?

ADT Trees: Functions

```
weight function
weight(nilTree) = 0
  | weight(fork(e,TL,TR)) = 1+weight(TL)+weight(TR)
```



What is the weight of the above tree?

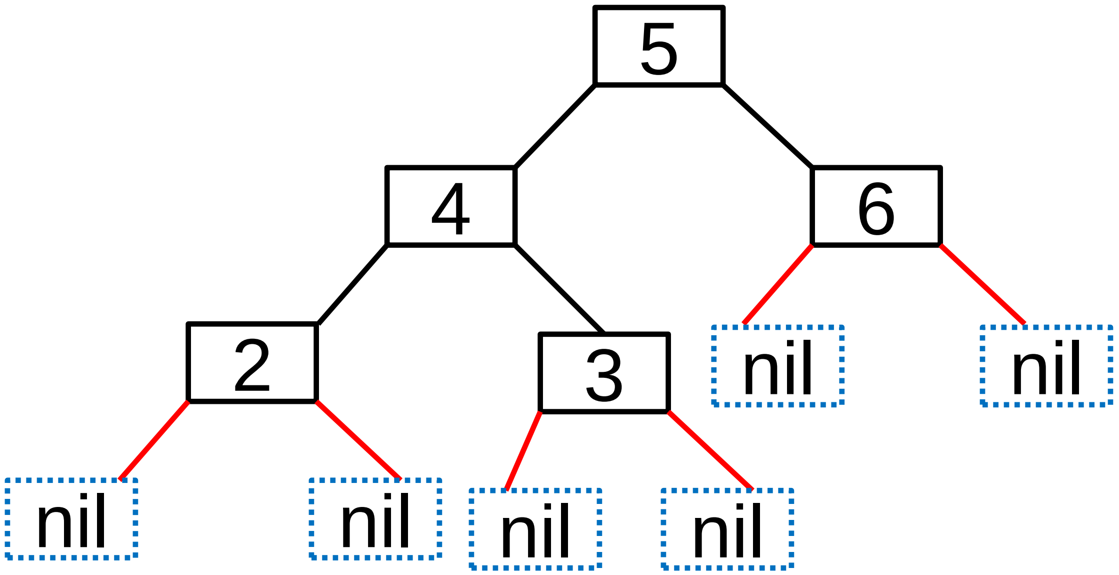
Recursive traversal of trees

- There are three classic ways of recursively traversing a tree or of visiting every one of its nodes once.
- In each of these, the left and right subtrees are visited recursively
- The **distinguishing feature** is when the element in the root is visited or processed.

Preorder or prefix traversal

- In a preorder or prefix traversal the root is visited first (pre) and then the left and right subtrees are traversed.

```
preorder(nilTree) = nil
| preorder(fork(e,TL,TR)) = print(e),preorder(TL),preorder(TR)
```



What is the preorder:

A. 23456

B. 54236

C. 54623

D. 24356

E. None of the above

Inorder or Infix traversal

- In an inorder or infix traversal, the left sub-tree is traversed then the root is visited and then the right sub-tree

```
inorder(nilTree) = nil
| inorder(fork(e,TL,TR)) = inorder(TL),print(e),inorder(TR)
```

What is the inorder:

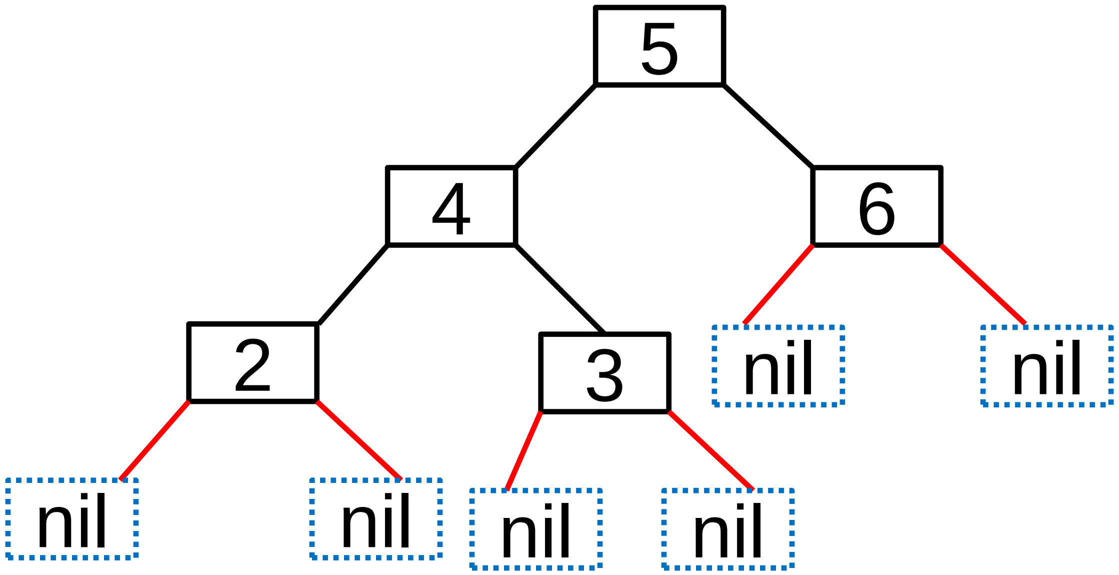
A. 23456

B. 54236

C. 54623

D. 24356

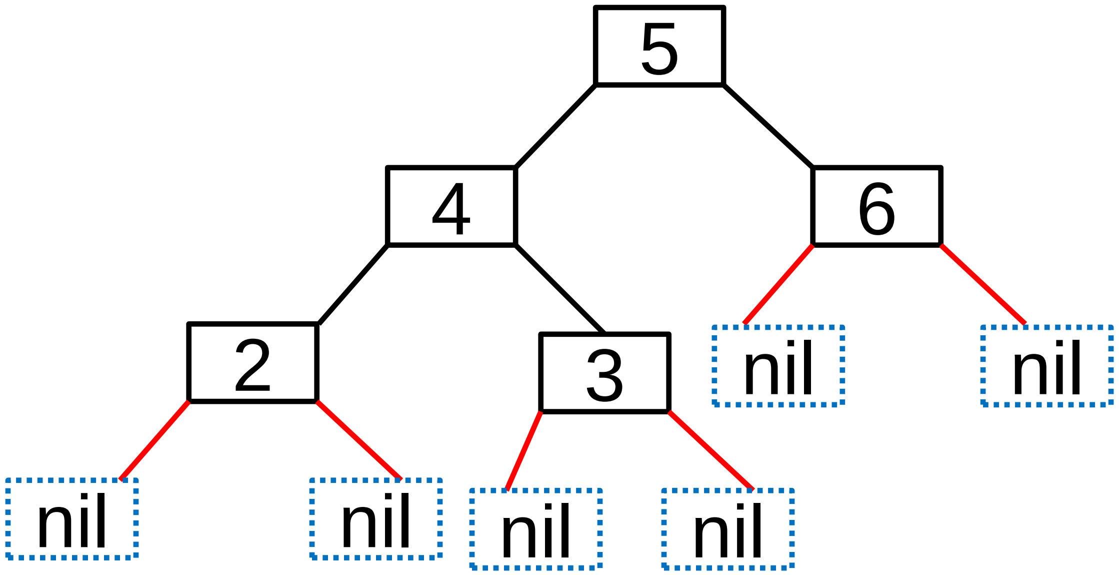
E. None of the above



Postorder or postfix traversal

- In a postorder or postfix traversal, the left and right sub-trees are traversed then the root is visited

```
postorder(nilTree) = nil
| postorder(fork(e,TL,TR))= postorder(TL),postorder(TR),print(e)
```



What is the inorder:

A. 23456

B. 54236

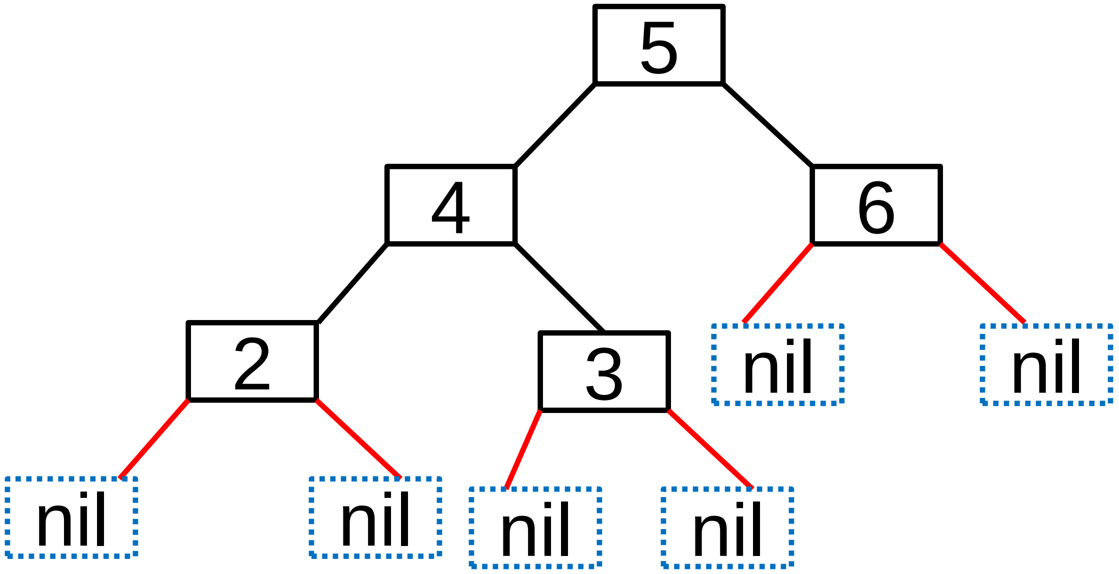
C. 54623

D. 24356

E. None of the above

Breadth-first traversal

- In a breadth-first traversal, root is visited then its children and then its grandchildren and so on. It is naturally an iterative traversal, hence, its recursive version is not given here.



One possible breadth-first order of the tree is 54623

Concluding Remarks

Take home message

- This unit demands your efforts from week 1
- ADTs provide flexibility and ease of proofs
- A proof is much stronger than a test

Things to do (this list is not exhaustive)

- Read more about content covered in this lecture
- If you do not understand computational complexity, study to develop some background (e.g., watch videos, read other online resources)

Coming Up Next

- Correctness, verification and analysis of algorithms
- Loop invariances, simple sorting and complexity issues