COMP2521 19T0
Week 3, Tuesday: Graphic Content (I)!

Jashank Jeremy

jashank.jeremy@unsw.edu.au

priority queues
graph fundamentals

Assignment 1: Textbuffer

Pitfalls and Pointers (I)

- Strings in C are pointers to arrays of `char`acters;
  following the last character is a `NUL` terminator: `'\0'`
  there won't be multiple `NUL` characters in a string

Assignment 1: Textbuffer

Pitfalls and Pointers (I)

- Strings in C are pointers to arrays of `char`acters;
  following the last character is a NUL terminator: `'\0'`
  there won't be multiple NUL characters in a string
- To store `"hello\n"`: **7 bytes** —
  ... `{'h', 'e', 'l', 'l', 'o', '\n', '\0'}`
  ... referring to the string `"\0"` is redundant

Assignment 1: Textbuffer

Pitfalls and Pointers (I)

- Strings in C are pointers to arrays of `char`acters;
  following the last character is a `NUL` terminator: `'\0'`
  there won't be multiple `NUL` characters in a string
- To store `"hello\n"`: **7 bytes** —
  … {`'h'`, `'e'`, `'l'`, `'l'`, `'o'`, `'\n'`, `'\0'`}
  … referring to the string `"\0"` is redundant
- `sizeof` is a *static* property; string length is a *dynamic* property.
  … in *(e.g.,)* `textbuffer_new`:
  … `sizeof text` = `sizeof (char *)` $= 4$
  … `sizeof *text` = `sizeof (char)` $= 1$
  … use *strlen(3)* or *strnlen(3)* or similar

Assignment 1: Textbuffer

Pitfalls and Pointers (II)

- Making a (heap-allocated, mutable) copy of a string?
  … *strdup(3)*, *strndup(3)* get it right — did you?

Assignment 1: Textbuffer

Pitfalls and Pointers (II)

- Making a (heap-allocated, mutable) copy of a string?
  … *strdup(3)*, *strndup(3)* get it right — did you?
- Splitting a string using *strsep(3)* or *strtok(3)*?
  … do you know what's going on?

Assignment 1: Textbuffer

Pitfalls and Pointers (II)

- Making a (heap-allocated, mutable) copy of a string?
  … *strdup(3)*, *strndup(3)* get it right — did you?
- Splitting a string using *strsep(3)* or *strtok(3)*?
  … do you know what's going on?
- HINT read the forum answers!
  … they tend to be filled with all kinds of useful wisdom

Assignment 1: Textbuffer

Pitfalls and Pointers (II)

- Making a (heap-allocated, mutable) copy of a string?
  … *strdup(3)*, *strndup(3)* get it right — did you?
- Splitting a string using *strsep(3)* or *strtok(3)*?
  … do you know what's going on?
- HINT read the forum answers!
  … they tend to be filled with all kinds of useful wisdom
- ANTI-HINT the challenge exercises are *challenging*
  … you will need to do your own reading and thinking
  … undo/redo hint: see week01thu lecture
  … diff hint: Levenshtein, but is it optimal?

Assignment 1: Textbuffer

Pitfalls and Pointers (II)

- Making a (heap-allocated, mutable) copy of a string?
  … *strdup(3)*, *strndup(3)* get it right — did you?
- Splitting a string using *strsep(3)* or *strtok(3)*?
  … do you know what's going on?
- HINT read the forum answers!
  … they tend to be filled with all kinds of useful wisdom
- ANTI-HINT the challenge exercises are *challenging*
  … you will need to do your own reading and thinking
  … undo/redo hint: see week01thu lecture
  … diff hint: Levenshtein, but is it optimal?
- Cryptic crossword hint: 'shaken players shift the load'.

Priority Queues

Priority

Not all queues are created equal...
ever been to a hospital?

Priority

Not all queues are created equal…
ever been to a hospital?

FIFO doesn't always cut it!
Sometimes, we need to process
in order of *key or priority*.

Priority Queues (PQueues or PQs)
provide this with
altered enqueue and dequeue.

Priority Queue

Operations

$\textsc{enpqueue} :: \mathcal{Q}' \to (\texttt{Item}, \texttt{prio}) \to \texttt{void}$
join or requeue an item with a priority to pqueue $\mathcal{Q}'$

$\textsc{depqueue} :: \mathcal{Q}' \to \texttt{Item}$
remove the item with highest priority from pqueue $\mathcal{Q}'$
(potentially including the priority; $\to (\texttt{Item}, \texttt{prio})$)

Priority Queue

<pqueue.h>

```c
typedef struct pqueue *PQueue;
typedef int pq_prio;

/** Create a new, empty PQueue. */
PQueue pqueue_new (void q);

/** Destroy a PQueue, releasing its resources. */
void pqueue_drop (PQueue pq);

/** Add an item with a priority to a PQueue. */
void pqueue_en (PQueue pq, Item it, pq_prio prio);

/** Remove the highest-priority item from a PQueue. */
Item pqueue_de (PQueue pq, pq_prio *prio);

/** Get the number of items in a PQueue. */
size_t pqueue_size (PQueue pq);
```

Priority Queue
Implementations

ordered array or ordered list:
insert $O(n)$, delete $O(1)$

unordered array or unordered list:
insert $O(1)$, delete $O(n)$

there must be a better way!

Heaps of Fun

Heaps are a good solution.
Commonly viewed as trees;
commonly implemented with arrays.

Two important properties:
heap order property,
a ‘top-to-bottom’ ordering of values;
complete tree property,
every level is as filled as possible

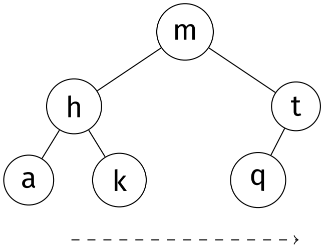
Heaps of Fun

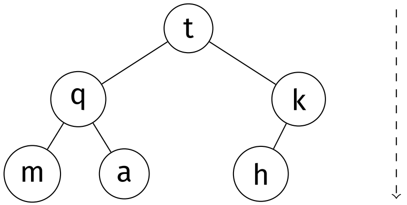
Heap-Order Property

Binary search trees have left-to-right ordering.

Heaps have a top-to-bottom ordering:
for all nodes, both subtrees are $\leq$ the root
(*i.e.*, the root contains the largest value)

Inserting $[\mathrm{m}, \mathrm{t}, \mathrm{h}, \mathrm{q}, \mathrm{a}, \mathrm{k}]$ into a BST and heap:

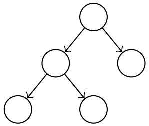


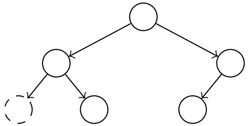


Heaps of Fun

Complete Tree Property

Heaps are *complete trees*:
every level is filled before adding nodes to the next level
nodes in a given level are filled left-to-right, with no breaks


complete

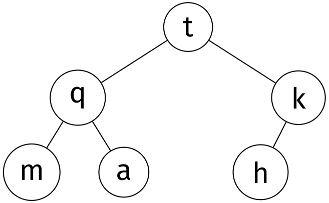

incomplete

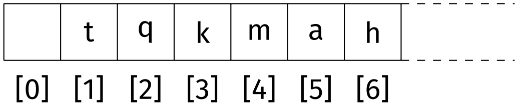
Heap Implementation

BSTs are typically implemented as linked data structures.

Heaps *can* be implemented as linked structures...
but are more commonly implemented as arrays.
complete tree $\Rightarrow$ array implementation

$$\textsc{left}\,(i) := 2i \qquad \textsc{right}\,(i) := 2i + 1 \qquad \textsc{parent}\,(i) := i/2$$



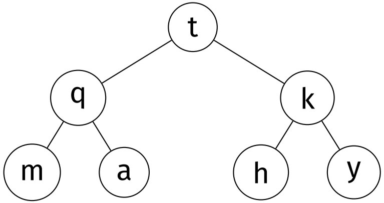


Heap Implementation

Insertion into an Array Heap (I)

Insertion is a two-step process:

1. add new element at the bottom-most, right-most position
   (to ensure it is still a complete tree)
2. reorganise values along the path to the root
   (to ensure it is still maintains heap order)

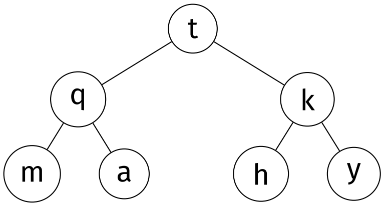


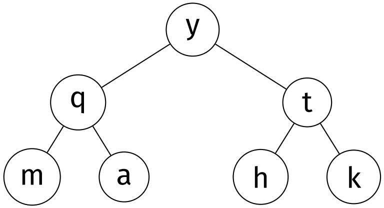
Heap Implementation

Insertion into an Array Heap (I)

Insertion is a two-step process:

1. add new element at the bottom-most, right-most position
   (to ensure it is still a complete tree)
2. reorganise values along the path to the root
   (to ensure it is still maintains heap order)





Heap Implementation

Insertion into an Array Heap (II)

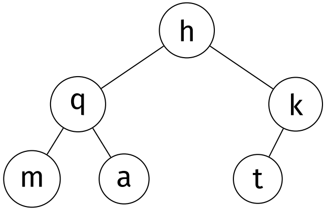
```
// move value at a[k] to correct position
void heap_fixup (Item a[], size_t k)
{
    while (k > 1 && item_cmp (a[k/2], a[k]) < 0) {
        swap (a, k, k/2);
        k /= 2; // integer division!
    }
}
```

Heap Implementation

Deletion from an Array Heap (I)

Deletion is a three-step process:

1. swap root value with bottom-most, right-most value
2. remove bottom-most, right-most value
   (to ensure it is still a complete tree)
3. reorganise values along path from root
   (to ensure it is still maintains heap order)

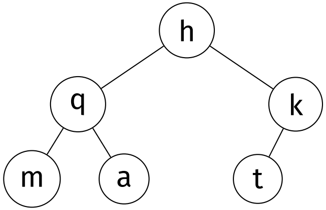


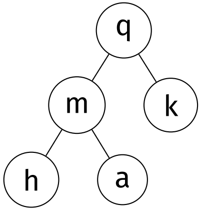
Heap Implementation

Deletion from an Array Heap (I)

Deletion is a three-step process:

1. swap root value with bottom-most, right-most value
2. remove bottom-most, right-most value
   (to ensure it is still a complete tree)
3. reorganise values along path from root
   (to ensure it is still maintains heap order)





Heap Implementation

Deletion from an Array Heap (II)

```
// move value at a[k] to correct position
void heap_fixdown (Item a[], size_t k)
{
    while (2 * k <= N) {
        size_t j = 2 * k; // choose greater child
        if (j < N && item_cmp (a[j], a[j+1]) < 0)
            j++;
        if (item_cmp (a[k], a[j]) >= 0)
            break;
        swap (a, k, j);
        k = j;
    }
}
```

Heap Complexity

Lots of work, surely?

height: always $\lfloor \log_2 n \rfloor$ (complete!)

insert: fixup is $O\left(\log_2 n\right)$
delete: fixdown is $O\left(\log_2 n\right)$

... worth it!

Now you try!

Exercise: Heaps of Fun!

Show the construction of the max-heap produced by inserting

$$[H, E, A, P, S, F, U, N]$$

Delete an item. What does the heap look like now?

Delete another item. What does the heap look like now?

PQueues

Graphs

Types of Graphs

Graph Terminology

Graph Fundamentals

Collections of Related Things

Up to this point, we've seen a few collection types...

lists: a *linear* sequence of items
each node knows about its next node
trees: a *branched* hierarchy of items
each node knows about its child node(s)

what if we want something more general?
...each node knows about its *related* nodes

Collections of Related Things

... Related Nodes? (I)

Many applications need to model relationships between items.

... on a map: cities, connected by roads
... on the Web: pages, connected by hyperlinks
... in a game: states, connected by legal moves
... in a social network: people, connected by friendships
... in scheduling: tasks, connected by constraints
... in circuits: components, connected by traces
... in networking: computers, connected by cables
... in programs: functions, connected by calls
... etc. etc. etc.

Collections of Related Things
... Related Nodes? (II)

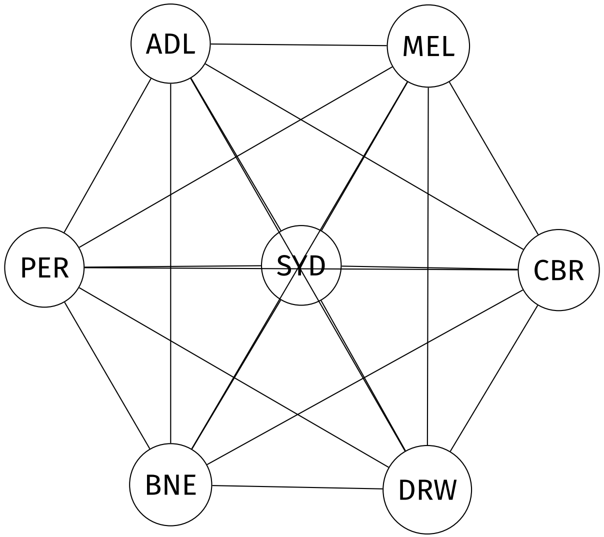
Questions we could answer with a graph:

- what items are connected? how?
- are the items fully connected?
- is there a way to get from A to B?
  what's the best way? what's the cheapest way?
- in general, what can we reach from A?
- is there a path that lets me visit all items?
- can we form a tree linking all vertices?
- are two graphs "equivalent"?

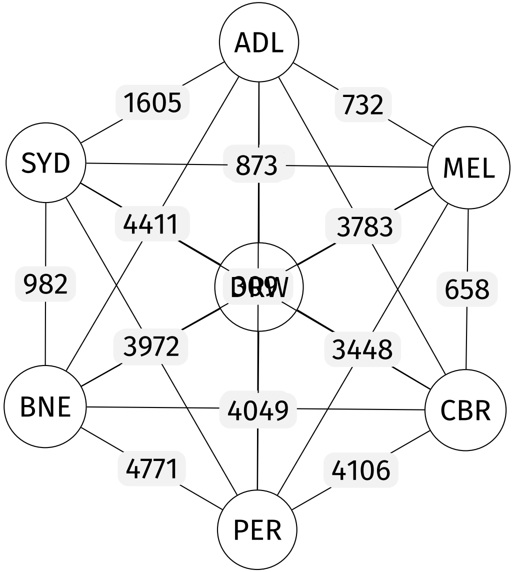
Road Distances

	ADL	BNE	CBR	DRW	MEL	PER	SYD
ADL	—	2055	1390	3051	732	2716	1605
BNE	2055	—	1291	3429	1671	4771	982
CBR	1390	1291	—	4441	658	4106	309
DRW	3051	3429	4441	—	3783	4049	4411
MEL	732	1671	658	3783	—	3448	873
PER	2716	4771	4106	4049	3448	—	3972
SYD	1605	982	309	4411	873	3972	—

Road Distances



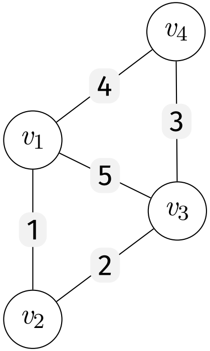
Road Distances



Graphs

A graph G is a set of vertices V and edges E.

$$E := \{(v, w) | v, w \in V, (v, w) \in V \times V\}$$

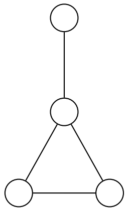


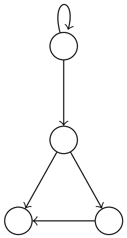
$$V = \{v_1, v_2, v_3, v_4\}$$

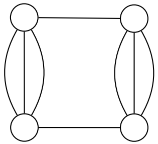
$$E = \left\{ \begin{array}{lcl} e_1 & := & (v_1, v_2), \\ e_2 & := & (v_2, v_3), \\ e_3 & := & (v_3, v_4), \\ e_4 & := & (v_1, v_4), \\ e_5 & := & (v_1, v_3) \end{array} \right\}$$

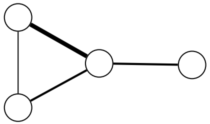
Graphs

Types of Graphs


undirected


directed


multigraph


weighted

Directed Graphs

If edges in a graph are directed,
the graph is a directed graph or digraph.

The edge $(v, w) \neq (w, v)$.
A digraph with V vertices can have at most V^2 edges.
Digraphs can have self loops ($v \to v$)

Unless otherwise specified,
graphs are undirected in this course.

Multigraphs and Weighted Graphs

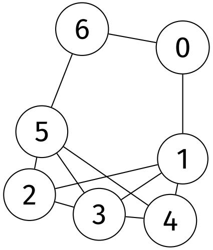
Multi-Graphs...
allow multiple edges between two vertices
(e.g., callgraphs; maps)

Weighted Graphs...
each edge has an associated weight
(e.g., maps; networks)

Simple Graphs

At this point,
we'll only consider simple graphs:

- a set of vertices
- a set of undirected edges
- no self loops
- no parallel edges



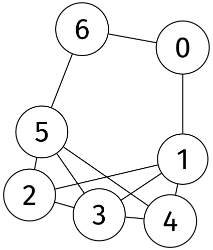
$|V| = 7; |E| = 11.$

How many edges can a
7-vertex simple graph have?

Simple Graphs

At this point,
we'll only consider simple graphs:

- a set of vertices
- a set of undirected edges
- no self loops
- no parallel edges



$|V| = 7; |E| = 11.$

How many edges can a
7-vertex simple graph have?

$7 \times (7-1)/2 = 21$

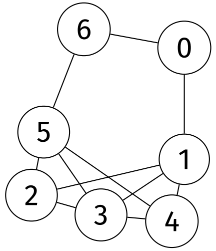
Graph Terminology (I)

For a simple graph:

$$|E| \leq (|V| \times (|V| - 1))/2$$

- if $|E|$ closer to $|V|^2$, *dense*
- if $|E|$ closer to $|V|$, *sparse*
- if $|E| = 0$, we have a set

These properties affect our choice of representation and algorithms.

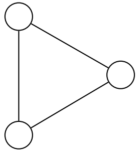


$|V| = 7$; $|E| = 11$.

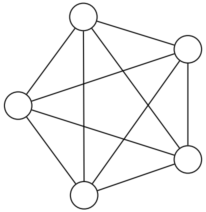
Graph Terminology
(II)

A complete graph is a graph where
every vertex is connected to all other vertices:

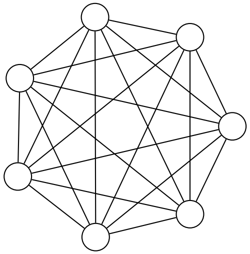
$$|E| = (|V| \times (|V| - 1))/2$$



K_3



K_5



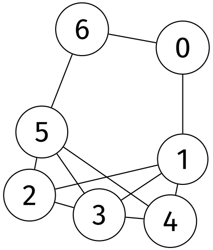
K_7

Graph Terminology (III)

A vertex v has degree $\deg(v)$
of the number of edges
incident on that vertex.

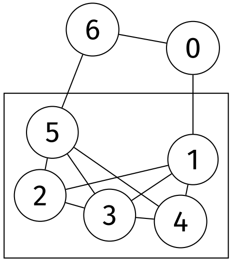
$\deg(v) = 0$ — an isolated vertex
$\deg(v) = 1$ — a pendant vertex

Two vertices v and w are adjacent
if an edge $e := (v, w)$ connects them;
we say e is incident on v and w



Graph Terminology
(IV)

A subgraph is a subset of vertices and associated edges

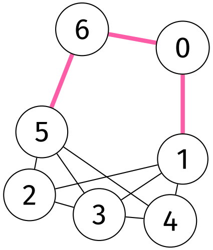


Graph Terminology
(V)

A path is
a sequence of
vertices and edges
... $1, 0, 6, 5$

a path is simple
if it has no repeating vertices

a path is a cycle
if it is simple *except*
for its first and last vertex,
which are the same.

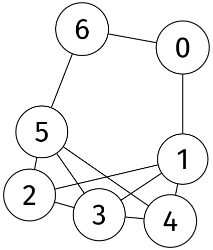


Graph Terminology
(VI)

A connected graph
has a path from every vertex
to every other vertex

A connected graph
with no cycles is a tree.

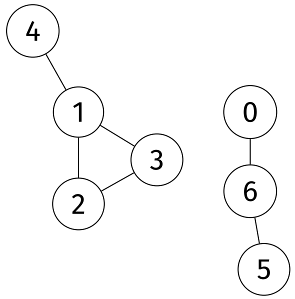
A tree has exactly one path
between each pair of vertices.



(not a tree)

Graph Terminology (VII)

A graph that is not connected consists of a set of **connected components**: maximally connected subgraphs

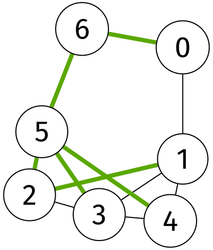


Graph Terminology (VIII)

A spanning tree of a graph
is a subgraph that
contains all its vertices
and is a single tree

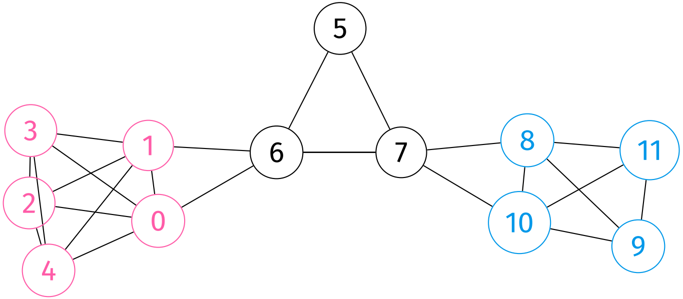
A spanning forest of a graph
is a subgraph that
contains all its vertices
and is a set of trees

There isn't necessarily *only* one
spanning tree/forest for a graph.



Graph Terminology
(IX)

A clique is a complete subgraph.


5
3
1
6
7
8
11
2
0
10
9
4