COMP20003
Algorithms and Data Structures
Dictionaries and Data Structures

Kris Ehinger
Department of Computing and Information Systems
University of Melbourne
Semester 2

Limited size

- We can overcome the limited size problem using dynamic memory allocation
- C library functions:
  - void *calloc(size_t nobj, size_t size)
  - void *realloc(void *p, size_t size)
  - also, of course void *malloc(size_t size)
  - All defined in `stdlib.h`

malloc(): size_t

- malloc(size_t size)
- size_t is:
  - an unsigned integer type
  - the type returned by the **sizeof** operator
  - widely used in the standard library (`stdlib`) to represent sizes
- *e.g.* malloc(**sizeof**(int))

malloc() example (part 1)

```
#define NUMBER 5
int main (argc, argv)
{
        int   var;
        var = NUMBER;
        printf("%d - %d\n", &var, var);
        return 0;
}

>a.out
134509940 - 5
```

malloc() example: (part 2)

```
#include<stdlib.h>
#include<stdio.h>
#define NUMBER 5
int main ()
{
   int* ptr;
   ptr = (int *)malloc(sizeof(int));
   *ptr = NUMBER; /* note '*' */
   printf("%d - %d\n", ptr, *ptr);
   return 0;
}
>a.out
134613280 - 5
```

Source: https://jdoodle.com/a/4fjm

COMP 20003 Algorithms and Data Structures 1-44

malloc():check return value

- Be aware: malloc() can fail!
  - If malloc() fails, it returns NULL
  - Never use a pointer to something where the memory allocation has failed!

```
int* B;
B = (int*) malloc( NUMBER * sizeof(int));
/* always check return value of malloc()*/
If( B == NULL )
{
    printf("malloc() error\n");
    exit(1);
}
```

→ write a function safemalloc() that does this

Getting memory for an array using malloc()

```
int A[NUMBER];
/**
 * while insertions < NUMBER array is OK
 * BUT… has a limit
 */
int* B;
/* always check return value of malloc()*/
if( (B = (int *) malloc( NUMBER * sizeof(int) )) == NULL )
{
        printf("malloc() error\n");
        exit(1);
}
/**
 * B can now be used like A
 * better to use calloc(NUMBER,sizeof(int))
 */
```

1-46

Getting memory for an array using calloc()

```
int* B;
/* always check return value of malloc()*/
if( (B = (int*) calloc( NUMBER, sizeof(int) )) == NULL )
{
        printf("calloc() error\n");
        exit(1);
}

/* B now comes with each slot initialized to 0 */
```

1-47

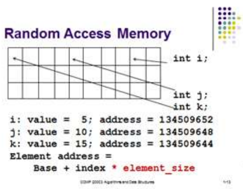
realloc()

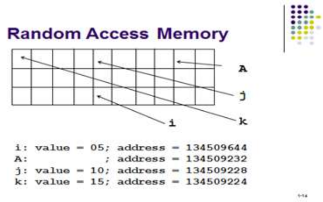
```
/**
 * as previously, used malloc(),calloc()
 * RESIZE when insertions == NUMBER
 */
B = realloc( B, ( NUMBER * 2 ) * sizeof(int) );
/* should also check realloc() for NULL*/

/* now initialize new part of array */
for( i = NUMBER; i < NUMBER * 2; i++ )
        B[ i ] = NULL;
/* now we have a bigger array, first half copied from the old B */
```

1-48

Details about malloc() and friends

- malloc**()** returns a pointer to a place in memory.
- Argument to malloc**()** specifies how much space to reserve in memory





What's a pointer?

- A pointer is an address in memory
- What is the output of this code?

```
int* ptr;
ptr = (int *)malloc(sizeof(int));
*ptr = 5;
printf("%d, %d", ptr, *ptr);
```

- Now what is the output of *this* code? Try it:https://jdoodle.com/a/4fP

```
int* ptr;
ptr = (int*) malloc(sizeof(int));
ptr = 5;
printf("%d, %d", ptr, *ptr);
```

- Int* p; or int *p; ? What the inventor of C++ thinks:
  - http://www.stroustrup.com/bs_faq2.html#whitespace

1-50

Details about malloc() and friends

- #include<stdlib.h>
- Read the documentation for fine points:
  - `malloc()` returns uninitialized space
  - `calloc()` returns space initialized to 0
  - `realloc(void *p, size_t size)` returns space where the start is copied from p and the rest is unintialized.
- Check return value of all memory alloc functions!

COMP 20003 Algorithms and Data Structures 1-51

malloc() and free()

- **`malloc()`** allocates memory.
- **`free()`** use to deallocate memory

```
void* ptr;
ptr = malloc(NUMBER_OF_BYTES);
/* do things until finished with the
   contents pointed to by ptr */
free(ptr);
```


More about Pointers

- For an excellent exposition of pointers in C, see the excellent tutorial by Ted Jensen:
  - LMS Resources → Pointers and Arrays in C

101 on Pointers

- Declaration: pointer * in C is an address
- Operators:

*	Value at Operator (dereferencing)	Gives Value stored at Particular address
&	Address of Operator	Gives Address of Variable

```
int k;
int* ptr;

k=5;
ptr = &k;
printf("%d", *ptr);
```

<5>

101 on Pointers

- A pointer in C is an address.
- When A is the name of an array, A is a pointer to the array, and
  - A[0] is equivalent to *(A+0),
  - A[5] is equivalent to *(A+5)…

Memory Allocation: Summary

- **malloc(),calloc(),and realloc()** return:
  - The (untyped) address of allocated memory;
  - *i.e.* a pointer to allocated memory.

```
/*allocates just enough room for an address */
struct node* ptr;
/* allocates enough room for the node */
ptr = (struct node*) malloc(sizeof(struct node));
```


Back to sorted arrays…

- Space limitations:
  - Can use **realloc()**.
  - Or can use linked list (sorted linked list).

We have discussed the strong and weak points of sorted vs. unsorted arrays as data structures for search

Linked lists: flexibility, but more overhead

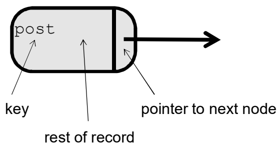
- In a linked list:
  - each item (or key) is located in an arbitrary place in memory,
  - with a link (pointer) to the next item
- Search Operations:
  - If unsorted, finding item is still $\Theta(n)$-time.
  - Once insertion point has been determined, easy to insert (or delete) a new item, by rearranging links.

Linked lists: flexibility, but more overhead

- Takes extra space for each item in the list.
- Takes extra time to allocate the memory for the node for each item.

The node

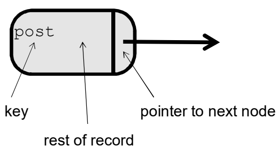
```
struct node{
    record r;
    struct node *next;
};
```



1-61

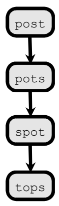
The node

```
typedef struct node{
    record r;
    struct node *next;
} node_t;
```



1-62

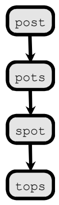
A linked list of nodes




```
struct node
{
   char* key;
   char* info;
   struct node* next;
};

struct node* newnode;

newnode = /*malloc space and put in the key and info */

/* suggested declaration style for beginner and intermediate */
```

1-63

A (sorted) linked list of nodes




```
/* record struct contains key and other info */

typedef struct node{
        record r;
        struct node* next;
        } node_t;

typedef node_t* node_ptr;

node_ptr   newnode;

/* more advanced style */
```

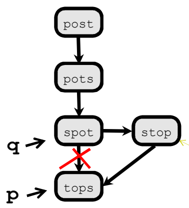

Traverse the list

```
p = listhead;
if(p!=NULL){ /* empty list */
   while( p->next !=NULL)
   {
      printf("%d\n", p->key);
      p = p->next;
   }
   printf("%d\n",p->key);
}
```


Traverse the list

```
p = listhead;
if(p==NULL)
{
   printf("List empty\n");
   return;
}

/* traverse and print key */
while( p->next !=NULL)
{
   printf("%d\n", p->key);
   p = p->next;
}
printf("%d\n",p->key);
```

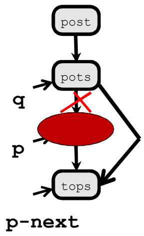

Inserting a new node into a sorted linked list




```
/* find correct place in list to insert */

/* how? */

/* then adjust pointers*/
q->next = newnode;
newnode->next=p;
```

newnode

Deleting a node




```
/* p points to the node we want to delete */
q->next = p->next;
free(p);
```

Fill the missing code: https://jdoodle.com/a/4fT

Linked lists: sorted *vs.* unsorted

- What are the advantages of keeping a linked list in sorted order?
- What are the disadvantages?

Search: Arrays vs. Linked Lists

- Sorted arrays:
  - Fast search (binary search), but
  - Slow insertion (keeping sorted order)
- Sorted array:
  - Fixed size, but
  - Can grow with realloc()
- Array needs (in general) only 1 memory allocation
  - Linked list needs many

Table of “running times”

	One Search	One Insert
Unsorted array		
Sorted array		
Unsorted linked list		
Sorted linked list		

Table of "running times"

	One Search	One Insert
Unsorted array	n	1
Sorted array	log n	n
Unsorted linked list	n	1
Sorted linked list	n	n

Exercise

How many operations are needed for m searches in a dictionary of n items?

	Each Insertion	Each Search	Build + Search
Unsorted array			
Sorted array			
Unsorted linked list			
Sorted linked list			

Practical complexity and algorithms

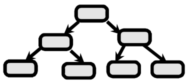
- O(1): Execute instructions once (or a few times), independent of input
  - Example: pick a lottery winner
- O(log n): keep splitting the input, and only operate on *one* section of the input.
  - Example:
- O(n): Execute instruction(s) once for each item:
  - Example:

Practical complexity and algorithms

- O(n log n): split the input repeatedly, and do something to *all* the segments
  - Example: Many sorting algorithms
- O(n^2): For each item, do something to all the others. (Nested loops.)
  - Example:
    - Note: getting slow for large data…
- O(n^3):
- O(2^n):

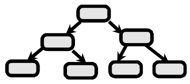
Breaking out of linearity

- Compare:
  - Linked list

  - Binary tree

- If we reliably know whether the desired item is in the left subtree or the right subtree, we could find it more quickly!

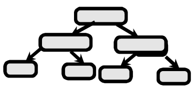
Breaking out of linearity

- Compare:
  - Linked list

  - Binary tree

- Note: for a complete binary tree, half the nodes are at the bottom level…

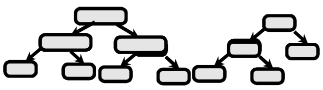
What is a complete binary tree?

- A binary tree is **complete** if **every level**, **except** possibly the **last**, is:
  - completely filled, and
  - **all nodes are as far left** as possible.



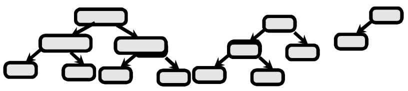
What is a complete binary tree?

- A binary tree is **complete** if **every level**, **except** possibly the **last**, is:
  - completely filled, and
  - **all nodes are as far left** as possible.



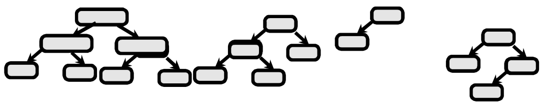
What is a complete binary tree?

- A binary tree is **complete** if **every level, except** possibly the **last**, is:
  - completely filled, and
  - **all nodes are as far left** as possible.

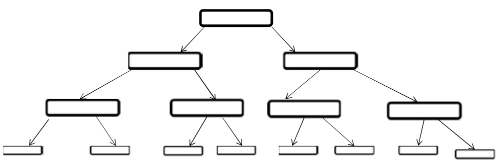


What is a complete binary tree?

- A binary tree is **complete** if **every level, except** possibly the **last**, is:
  - completely filled, and
  - **all nodes are as far left** as possible.

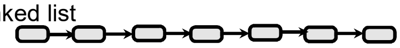
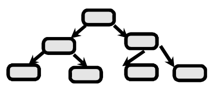


complete binary tree

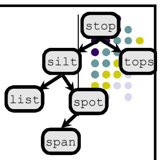


A complete binary tree of n nodes has depth approximately $\log_2 n$

Breaking out of linearity

- Compare:
  - Linked list 
  - Binary tree 
- As we will see, both insertion and search are O(log *n*) operations.

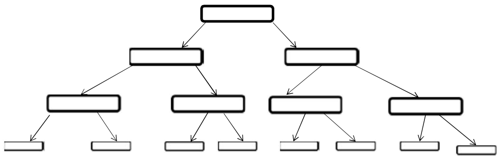
How does a binary search tree work?



- In a sorted linked list:
  - **`next`** links to a record with a key ≥ this one.
- In a BST
  - **`left`** links to items with key < current key
  - **`right`** links to items with key ≥ current key.
- Insert node with key **`span`** in this tree.

Looking at a complete binary tree

Nodes



2^0

2^1

2^2

2^3

….

2^{logn-1}

For a complete binary tree of n nodes, to get to the bottom takes not more than $\log_2 n$ key comparisons

linked list, to get to the end, we need how many comparisons?

Binary tree exercizes

- Put the following numeric keys into a bst:
  - 45, 37, 86, 90, 50, 16, 37
  - How long (how many key comparisons) does it take to search for key=5?

https://www.cs.usfca.edu/~galles/visualization/BST.html

- Put the following numeric keys into a bst:
  - 90, 86, 50, 45, 37, 32, 16
  - How long does it take to search for key=5?

Binary tree exercizes

- Put the following numeric keys into a bst:
  - 45, 37, 86, 90, 50, 16, 37
  - How long (how many key comparisons) does it take to search for key=5?

- Put the following numeric keys into a bst:
  - 90, 86, 50, 45, 37, 32, 16
  - How long does it take to search for key=5?

Best case run time in bst: Perfectly balanced tree

- Best case for BST: perfectly balanced
- Height of tree with n items: $\log_2 n$
- Path from root to any node:
  - Maximum length: $\log_2 n$
  - Average length: $\log_2 n$
- Insertion/search/deletion are all O(log n) for a well-balanced tree

Worst case run time in bst: Stick

- Worst case for BST: a stick.
  - e.g. when items are inserted in sorted order.
  - The BST degenerates to a linked list!
- Height of tree with n items: n
- Path from root to any node:
  - maximum length: n
  - average length: n/2
- Insertion/search/deletion are O(n)!

Binary search trees

- Deletion?

COMP 20003 Algorithms and Data Structures 1-90

Something to think about

- Why don't we just randomize the order of the items we insert into the binary search tree, to prevent worst case behavior?

COMP 20003 Algorithms and Data Structures 1-91