COMSM1201
Algorithms & Data Structures

Dr. Neill Campbell

Room 3.14 MVB

Neill.Campbell@bristol.ac.uk

Simple Recursion

- When a function calls itself, this is known as recursion.

- This is an important theme in Computer Science that crops up time & time again.

- Can sometimes lead to very simple and elegant programs.

Fibonacci Sequences

A well known example of a recursive function is the Fibonacci sequence. The first term is 1, the second term is 1 and each successive term is defined to be the sum of the two previous terms, i.e. :

```
fib(1) is 1
fib(2) is 1
fib(n) is fib(n-1)+fib(n-2)

1,1,2,3,5,8,13,21, ...
```

Iteration & Fibonacci Sequences

```
#include <stdio.h>

int fibon(int n);

int main(void)
{

   int i;
   for(i=1; i<20; i++){
      printf("%d = %d\n", i, fibon(i));
   }

   return 0;

}
```

Iteration & Fibonacci Sequences

```
int fibon(int n)
{
   int i, a, b, temp;

   if(n == 1) return 1;
   if(n == 2) return 1;

   a = 1; b = 1;
   for(i=3; i<=n; i++){
      temp = a;
      a = b;
      b = temp + a;
   }
   return b;
}
```

Recursive Fibonacci Sequence

```
int fibon(int n)
{

   if(n == 1) return 1;
   if(n == 2) return 1;

   return( fibon(n-1)+fibon(n-2));

}
```

Iterative String Reverse

```
#include <stdio.h>
#include <string.h>

#define SWAP(A,B)
        {char temp; temp=A;A=B;B=temp;}

void Reverse_String(char *s, int n);

int main(void)
{

   char str[] = "Hello World";

   Reverse_String(str, strlen(str)-1);
   printf("%s\n", str);

}

/* Iterative Inplace String Reverse */
void Reverse_String(char *s, int n)
{

   int i, j;

   for(i=0, j=n-1; i<j; i++, j--){
       SWAP(s[i], s[j]);
   }

}
```

Recursive String Reverse

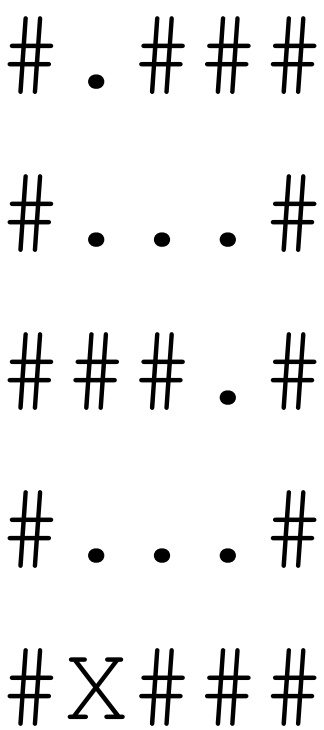
```
#include <stdio.h>
#include <string.h>

#define SWAP(A,B) {char temp; temp=A;A=B;B=temp;}

void Reverse_String(char *s, int start, int end);

int main(void)
{

   char str[]= "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

   Reverse_String(str, 0, strlen(str)-1);
   printf("%s\n", str);

}

/* RECURSIVE : Inplace String Reverse */
void Reverse_String(char *s, int start, int end)
{

   if(start < end){
   /* Debugging Info */
   printf("Reverse String -->%s<--\n", s);
      /* Here's where all the work is done */
      SWAP(s[start], s[end]);
      Reverse_String(s, start+1, end-1);
   }

}
```

Output

```
Reverse String -->ABCDEFGHIJKLMNOPQRSTUVWXYZ<--
Reverse String -->ZBCDEFGHIJKLMNOPQRSTUVWXYA<--
Reverse String -->ZYCDEFGHIJKLMNOPQRSTUVWXBA<--
Reverse String -->ZYXDEFGHIJKLMNOPQRSTUVWCBA<--
Reverse String -->ZYXWEFGHIJKLMNOPQRSTUVDCBA<--
Reverse String -->ZYXWVFGHIJKLMNOPQRSTUEDCBA<--
Reverse String -->ZYXWVUGHIJKLMNOPQRSTFEDCBA<--
Reverse String -->ZYXWVUTHIJKLMNOPQRSGFEDCBA<--
Reverse String -->ZYXWVUTSIJKLMNOPQRHGFEDCBA<--
Reverse String -->ZYXWVUTSRJKLMNOPQIHGFEDCBA<--
Reverse String -->ZYXWVUTSRQKLMNOPJIHGFEDCBA<--
Reverse String -->ZYXWVUTSRQPLMNOKJIHGFEDCBA<--
Reverse String -->ZYXWVUTSRQPOMNLKJIHGFEDCBA<--
ZYXWVUTSRQPONMLKJIHGFEDCBA
```

The Ubiquitous Maze

The correct route through a maze can often be obtained via recursive rather than iterative methods.



```
int explore(int x, int y, char mz[YS][XS])
{

   if  mz[y][x] is exit return 1;

   Mark mz[y][x] as 'wall'
        /* so we dont return here */

   if we can go up :
      if(explore(x, y+1, mz)) return 1

   if we can go right :
      if(explore(x+1, y, mz)) return 1

   Do left & down in a similar manner

   return 0; /* Failed to find route */

}
```

Permuting

```
/* Borrowed from e.g.
   http://www.geeksforgeeks.org
*/
#include <stdio.h>
#include <string.h>

int main()
{
    char str[] = "ABC";
    int n = strlen(str);
    permute(str, 0, n-1);
    return 0;
}

void permute(char *a, int l, int r)
{
   int i;
   if (l == r){
     printf("%s\n", a);
     return;
   }
   for (i = l; i <= r; i++){
      swap((a+l), (a+i));
      permute(a, l+1, r);
      swap((a+l), (a+i)); /*backtrack*/
   }
}
```

Simple Searching

- The need to search an array for a particular value is a common problem.

- This is used to delete names from a mailing list, or upgrading the salary of an employee etc.

- The simplest method for searching is called the sequential search.

- Simply move through the array from beginning to end, stopping when you have found the value you require.

Sequential Search 1

```
#include <stdio.h>
#include <strings.h>

#define NUMPEOPLE 6

struct key {
   char *name;
   int age;
};

typedef struct key Key;

int FindAge(char *name, Key *l, int n);

int main(void)
{

   int i, j;
   Key a[NUMPEOPLE] = {
         {"Ackerby", 21},
         {"Bloggs",  25},
         {"Chumley", 26},
         {"Dalton",  25},
         {"Eggson",  22},
         {"Fulton",  41}  };

   i = FindAge("Eggson",   a, NUMPEOPLE));
   j = FindAge("Campbell", a, NUMPEOPLE));
   printf("%d %d\n", i, j);

   return 0;

}
```

Sequential Search 2

```
int FindAge(char *name, Key *l, int n)
{

   int j;

   for(j=0; j<n; j++){
      if(strcmp(name, l[j].name) == 0){
         return l[j].age;
      }
   }

   return -1;

}
```

Ordered Sequential Search

- **If** we know that the array is ordered on the basis of names, then we can stop searching once the search key is alphabetically greater than the item at the current position in the list.

- In large lists this can save an enormous amount of time.

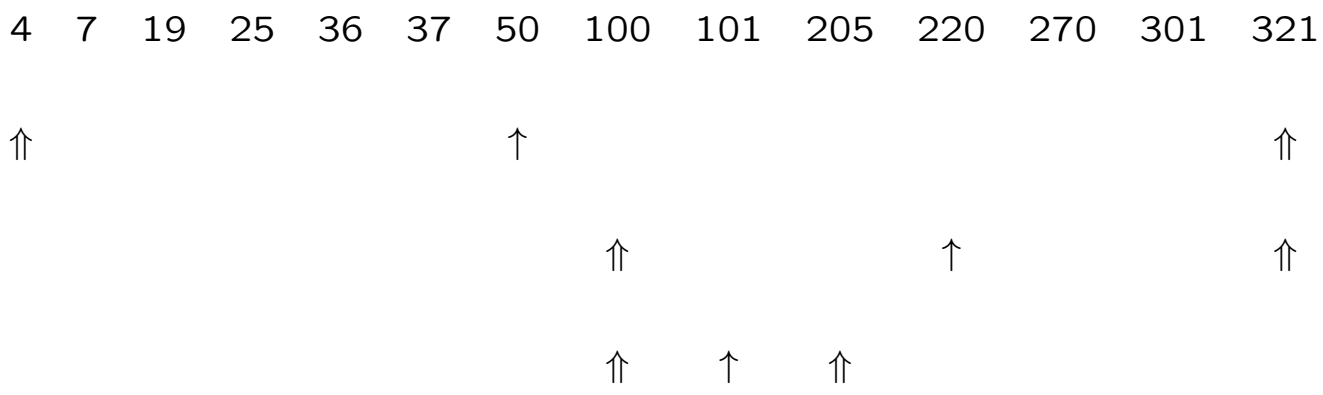
```
int FindAge(char *name, Key *l, int n)
{
   int j, m;

   for(j=0; j<n; j++){
      m = strcmp(name, l[j].name);
      if(m == 0)
         return l[j].age;
      if(m < 0)
         return -1;
   }
   return -1;
}
```

Binary Search for “101”

- Searching small lists doesn’t require much computation time.

- However, as lists get longer (e.g. phone directories), sequential searching becomes extremely inefficient.

- A binary search consists of examining the middle element of the array to see if it has the desired value. If not, then half the array may be discarded for the next search.



Binary Search

```
#include <stdio.h>

int isin(int k, int *a, int n);

int main(void)
{

   int a[] = {4,7,19,25,36,37,50,100,
              101,205,220,271,301,321};

   if(isin(101, a, 14) > 0){
      printf("Found It !\n");
   }
   else
      printf("Not in List\n");

   return 0;

}
```

Binary Search

```
int isin(int k, int *a, int n)
{

   int l, r, m;
   l = 0;
   r = n - 1;
   while(l <= r){
      m = (l+r)/2;
      if(k == a[m])
           return m;
      else{
         if (k > a[m])
              l = m + 1;
         else
              r = m- 1;
      }
   }

   return -1;

}
```

Recursive Binary Search

- Much of the processing in the previous function was controlled by a while() loop.

- We now know how to replace this by careful use of recursion.

```
int RecBin(int k, int *a, int l, int r)
{

   int m;

   if(l > r) return -1;
   m = (l+r)/2;
   if(k == a[m])
      return m;
   else{
      if (k > a[m])
         return RecBin(k, a, m+1, r);
      else
         return RecBin(k, a, l, m-1);
   }

}
```

Interpolation Search

- When we look for a name in a phone book, we dont start in the middle. We make an educated guess as to where to start based on the name of the person to be searched for.

- This idea has lead to the interpolation search.

- In binary searching, we simply used the middle of an ordered list as a best guess as to where to begin the search.

- Now we use an interpolation involving the key, the start of the list and the end.

$$i = (k - l[0])/(l[n-1] - l[0]) * n$$

- when searching for '15' :

0 4 5 9 10 12 15 20

⇑

$$i = (15 - 0)/(20 - 0) * 8$$

Algorithmic Complexity

- Searching and sorting algorithms have a complexity associated with them, called big-O.

- This complexity indicates how, for n numbers, performance deteriorates when n changes.

- Sequential Search : **O(n)**

- Binary Search : **O(log n)**

- Interpolation Search : **O(log log n)**

- We'll discuss the dream of a **O(1)** search later in "Hashing".

Execution Timing

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define CSEC (double)(CLOCKS_PER_SEC)

int main(void)
{

   clock_t c1, c2;
   int i, j;

   c1 = clock();
   for(i=0; i<10000000; i++)
        j = i * 2;
   c2 = clock();
   printf("%f\n", (double)(c2-c1)/CSEC);
   return 0;

}
```

This code on my old Sun UltraSparc took:

- 0.07 seconds using an optimising compiler.
- 1.28 seconds using a non-optimising compiler.
- 0.43 seconds on an old Mac.

Linked Data Structures

- Linked data representations are useful when:
  - It is difficult to predict the size and the shape of the data structures in advance.
  - We need to efficiently insert and delete elements.
- To create linked data representations we use pointers to connect separate blocks of storage together. If a given block contains a pointer to a second block, we can follow this pointer there.
- By following pointers one after another, we can travel right along the structure.

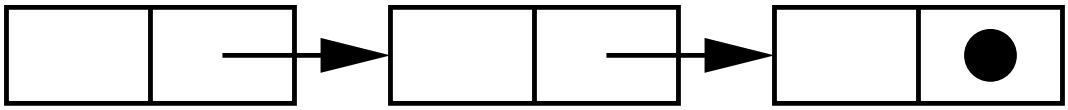
Types of Structures

At the highest level of abstraction, data types that we can represent using dynamic structures include:

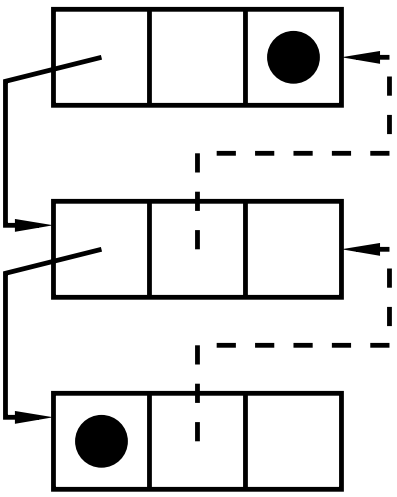
- Lists
- Stacks
- Queues
- Sets
- Graphs
- Trees

Linked Lists

- Linear Linked Lists:

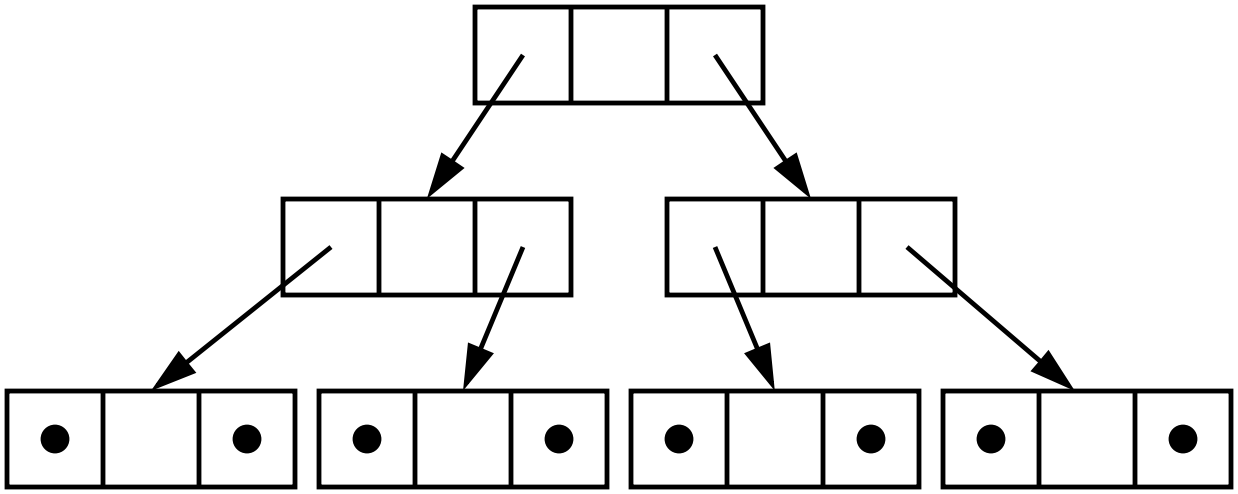


- Doubly Linked Lists:

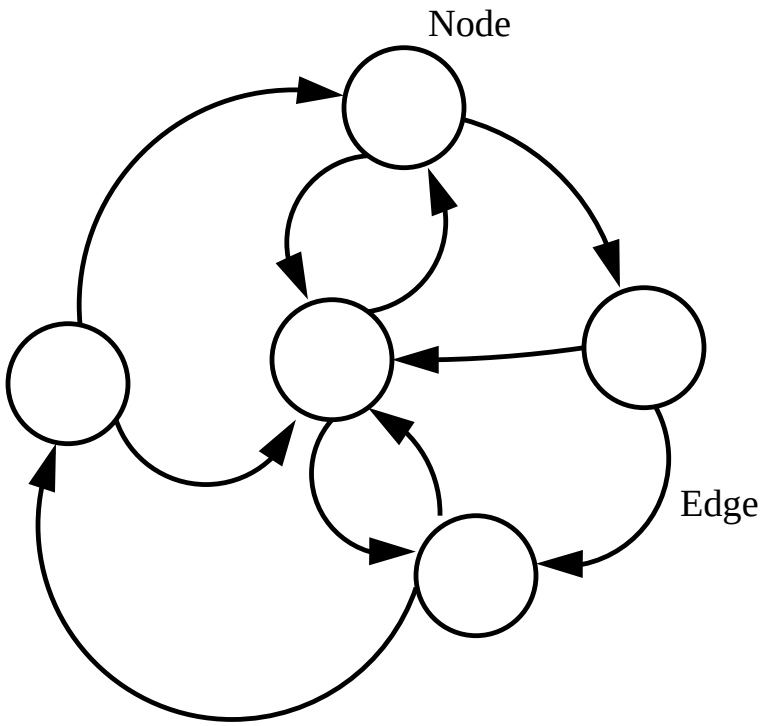


Trees & Graphs

Binary Trees:

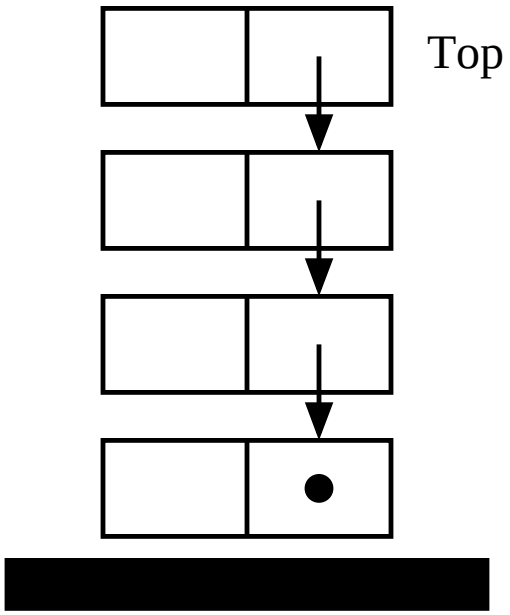


Unidirectional Graph:



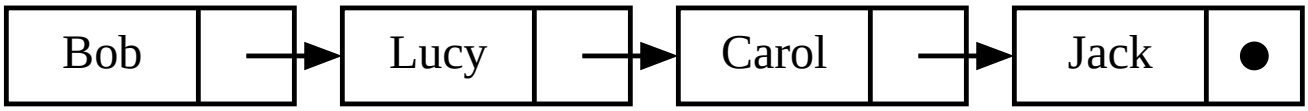
Stacks

Push-Down Stack:



Linear Linked Lists

A list of names:




```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXNAME 20

struct node{
   char name[MAXNAME];
   struct node *next;
};

typedef struct node Node;

Node *AllocateNode(char *s);
void PrintList(Node *l);
```

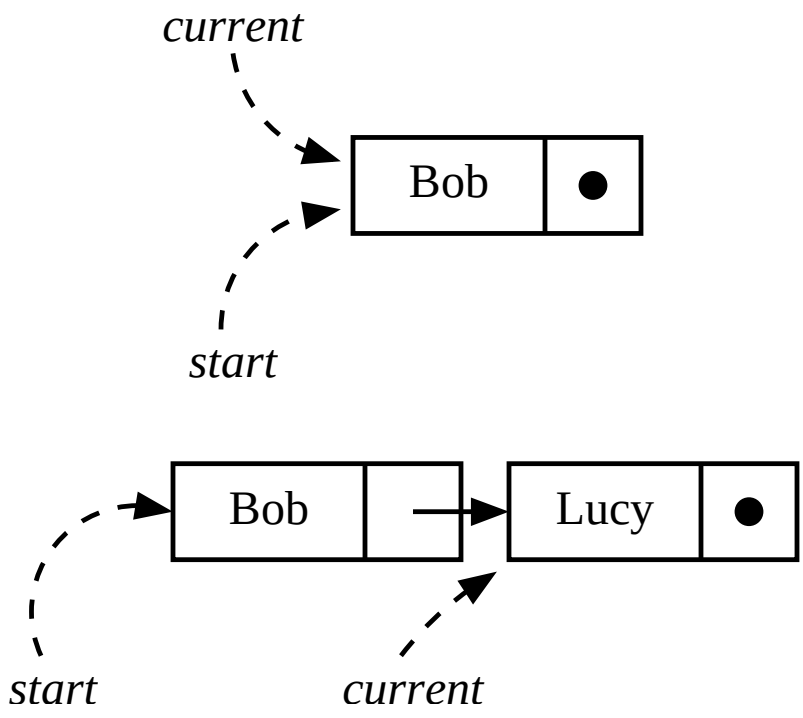
Linked List Program

```
int main(void)
{

   char name[MAXNAME];
   Node *start, *current;

   printf("Enter the first name : ");
   if(scanf("%s", name) == 1){
      start = current = AllocateNode(name);
   }
   else return 1;

   printf("Enter more names : ");
   while(scanf("%s", name) == 1){
      current->next = AllocateNode(name);
      current = current->next;
   }
   PrintList(start);
   return 0;

}
```

Linked List Program

```
Node *AllocateNode(char *s)
{

   Node *p;

   p = (Node *)malloc(sizeof(Node));

   if(p==NULL){
      printf("Cannot Allocate Node\n");
      exit(2);
   }

   strcpy(p->name, s);
   p->next = NULL;
   return p;

}
```

Building the List



Printing the List

```
void PrintList(Node *l)
{

   printf("\n");
   do{
      printf("Name : %s\n", l->name);
      l = l ->next;
   }while(l != NULL);
   printf("END\n");

}
```

Searching a List

```
Node *InList(Node *n, char *s)
{

   do{
       if(strcmp(n->name, s) == 0){
           return n;
       }
      n = n->next;
   }while(n != NULL);

   return NULL;

}
```

When the following is run:

```
   printf("%p\n", InList(start, "Jack"));
   printf("%p\n", InList(start, "Bob"));
   printf("%p\n", InList(start, "Joe"));
```

The following is returned:

```
20d58
20cf8
0
```

Recursive List Printing

```
void PrintList(Node *l)
{

   /* Recursive Base-Case */
   if(l == NULL) return;

   printf("Name : %s\n", l->name);
   PrintList(l ->next);

}
```

Recursive List Searching

```
Node *InList(Node *n, char *s)
{

   /* Recursive Base-Case */
   if(n == NULL) return NULL;

   if(strcmp(n->name, s) == 0) return n;
   return InList(n->next, s);

}
```

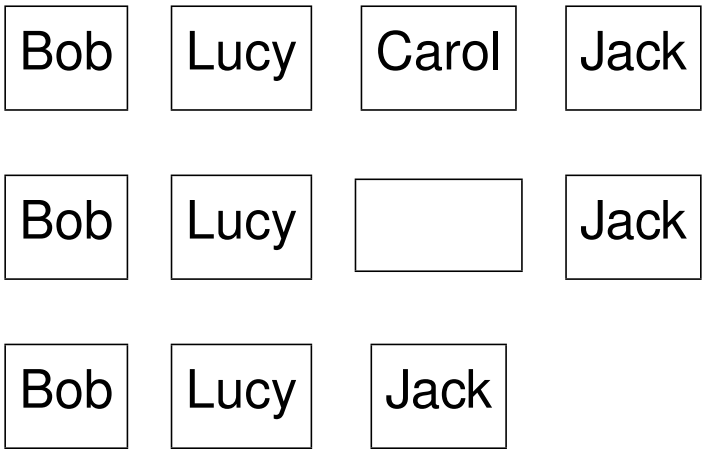
Arrays of Structures

Why are dynamic structures better than arrays ?

If we wanted to delete an element from an array of structures we needed to:

- Find the element to be deleted.

- Shuffle all the elements along one.



Sequential Lists

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXNAME 20

struct node{
   char name[MAXNAME];
};

typedef struct node Node;

int FillNode(Node l[], char *s, int n);
void PrintList(Node l[], int size);

int main(void)
{
   char name[MAXNAME];
   int current;
   Node list[100];

   printf("Enter the first name : ");
   if(scanf("%s", name) == 1)
      current = FillNode(list, name, 0);
   else return 1;

   printf("Enter more names : ");
   while(scanf("%s", name) == 1){
      current = FillNode(list, name, current+1);
   }
   ++current;
   PrintList(list, current);
   return 0;
}
```

Sequential Lists

```
int FillNode(Node l[], char *s, int n)
{
   strcpy(l[n].name, s);
   return n;
}

void PrintList(Node l[], int size)
{

   int i;

   for(i=0; i<size; i++){
      printf("Name : %s\n", l[i].name);
   }
   printf("END\n");

}
```

Deleting Elements is Expensive

```
void DelElem(Node l[], int n, int *size)
{
   int i;

   for(i=n+1; i<*size; i++){
        l[i-1] = l[i];
   }
   *size = *size - 1;
}
```

Search/Delete Linked List

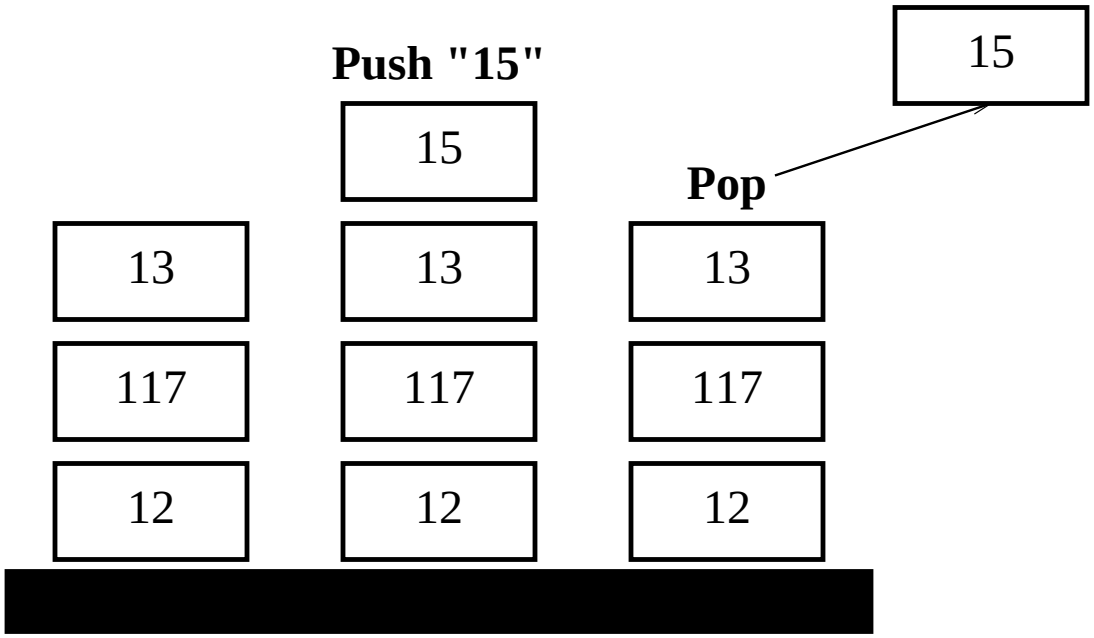
```
Node *DeleteNode(Node *start, char *s)
{

   Node *prev, *l;

   /* 1st in list ? */
   if(!strcmp(start->name, s))
      return start->next;

   l = start;
   do{
      prev = l;
      l = l->next;
      if(strcmp(l->name, s) == 0){
         prev->next = l->next;
         return start;
      }
   }while(l != NULL);

   return start;

}
```

Sequential Vs. Linked Lists

Task	Sequential Array	Linked List
Inserting a new 1^{st} element	O(n)	O(1)
Deleting the last element	O(1)	O(n)
Replacing the i^{th} element	O(1)	O(n)
Deleting the i^{th} element	O(n)	O(1)

Stacks

LIFO (Last in, First out):



- Operations include `push` and `pop`.

- In the C run-time system, function calls are implemented using stacks.

- Most recursive algorithms can be re-written using stacks instead.

Sequential Stacks

```
#include <stdio.h>
#include <assert.h>

#define STACKSIZE 200

struct stackelem{
   int i;
};
typedef struct stackelem Elem;

struct thestack{
   Elem stk[STACKSIZE];
   int top;
};
typedef struct thestack Stack;

void InitialiseStack(Stack *s);
void Push(Stack *s, int n);
int Pop(Stack *s);
```

Sequential Stacks

```
int main(void)
{

   Stack s;
   InitialiseStack(&s);

   Push(&s, 12);
   Push(&s, 117);
   Push(&s, 13);
   Push(&s, 15);
   printf("%d\n", Pop(&s));
   printf("%d\n", Pop(&s));
   printf("%d\n", Pop(&s));
   printf("%d\n", Pop(&s));
   printf("%d\n", Pop(&s));

   return 0;

}
```

Sequential Stacks

```
void InitialiseStack(Stack *s)
{
   s->top = 0;
}

void Push(Stack *s, int n)
{
   /* Make sure stack doesnt overflow */
   assert(s->top < STACKSIZE);
   s->stk[s->top].i = n;
   s->top = s->top + 1;
}

int Pop(Stack *s)
{

   /* Can't pop empty stack (underflow) */
   assert(s->top > 0);
   s->top = s->top - 1;
   return s->stk[s->top].i;

}
```

Running the Code

```
15
13
117
12
stack.c:64: failed assertion `s->top > 0'
Abort
```

Once we are sure that the code is working, we could speed it up by switching off all the checking done by assert(). This is done by defining NDEBUG before the assert include :

```
#define NDEBUG
#include <assert.h>
```

Abstract Data Types

If the exact definition of the stack structure and the functions which access it e.g. `InitialiaseStack()`, `Pop()` and `Push()` are all stored in other .c & .h files, then the details of the stack module implementation becomes unimportant to the programmer.

The abstraction of data types is one of the main themes of this course.

Dynamic Stacks

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct stackelem{
   int i;
   struct stackelem *prev;
};
typedef struct stackelem Elem;

struct thestack{
   Elem *tp;
};
typedef struct thestack Stack;

void InitialiseStack(Stack *s);
void Push(Stack *s, int n);
int Pop(Stack *s);
```

Dynamic Stacks

```
/* main is eactly the same as before */
int main(void)
{

   Stack s;
   InitialiseStack(&s);

   Push(&s, 12);
   Push(&s, 117);
   Push(&s, 13);
   Push(&s, 15);
   printf("%d\n", Pop(&s));
   printf("%d\n", Pop(&s));
   printf("%d\n", Pop(&s));
   printf("%d\n", Pop(&s));
   printf("%d\n", Pop(&s));

   return 0;

}
```

Dynamic Stacks

```
void InitialiseStack(Stack *s)
{
   s->tp = (Elem *)calloc(1, sizeof(Elem));
   s->tp->prev = NULL;
}

void Push(Stack *s, int n)
{
   Elem *e;
   e = (Elem *)calloc(1, sizeof(Elem));
   e->prev = s->tp;
   s->tp->i = n;
   s->tp = e;
}

int Pop(Stack *s)
{

   s->tp = s->tp->prev;
   assert(s->tp != NULL);
   return s->tp->i;

}
```

Using Stacks to Avoid Recursion

We've seen `StringReverse()` coded in an iterative (in-place) manner and also in a recursive manner.

Now we shall use a stack to avoid recursion.

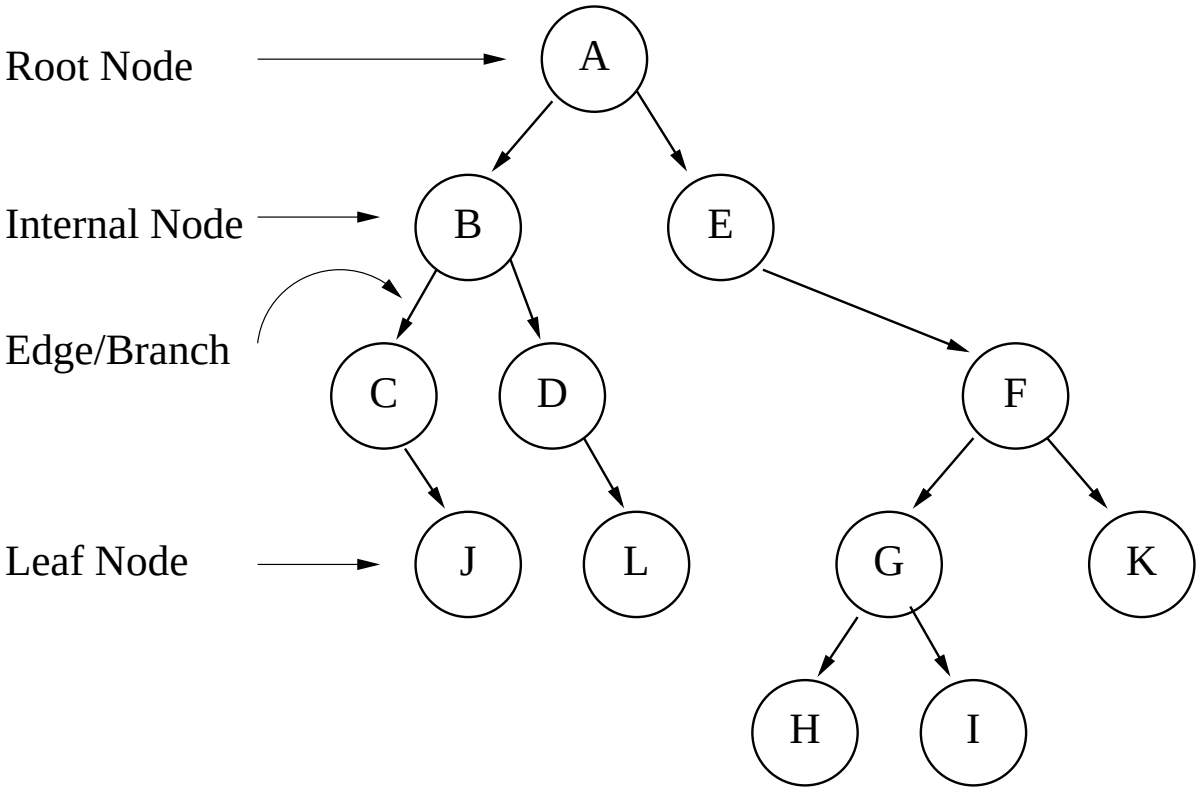
```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct stackelem{
   int i;
   struct stackelem *prev;
};
typedef struct stackelem Elem;

struct thestack{
   Elem *tp;
};
typedef struct thestack Stack;

void StringReverse(char *s);
void InitialiseStack(Stack *s);
void Push(Stack *s, int n);
int Pop(Stack *s);
```

Stack StringReverse

```
int main(void)
{

   char str[] = "Hello World";
   StringReverse(str);
   printf("%s\n", str);
   return 0;

}

void StringReverse(char *str)
{
   char *k;
   int i;
   int l = 0;
   Stack s;
   InitialiseStack(&s);

   k = str;
   while(*k != '\0'){
      Push(&s, (int)*k);
      k++;
      l++;
   }

   k = str;
   for(i=0; i<l; i++){
      *k = Pop(&s);
      k++;
   }
}
```

Binary Trees

- Binary trees are used extensively in computer science
- Game Trees
- Searching
- Sorting



Nomenclature

- Trees drawn upside-down !
- Ancestor relationships: A is the parent of E
- Can refer to left and right children
- In a tree, there is only one path from the root to any child
- A node with no children is a leaf
- Most trees need to be created dynamically
- Empty subtrees are set to NULL

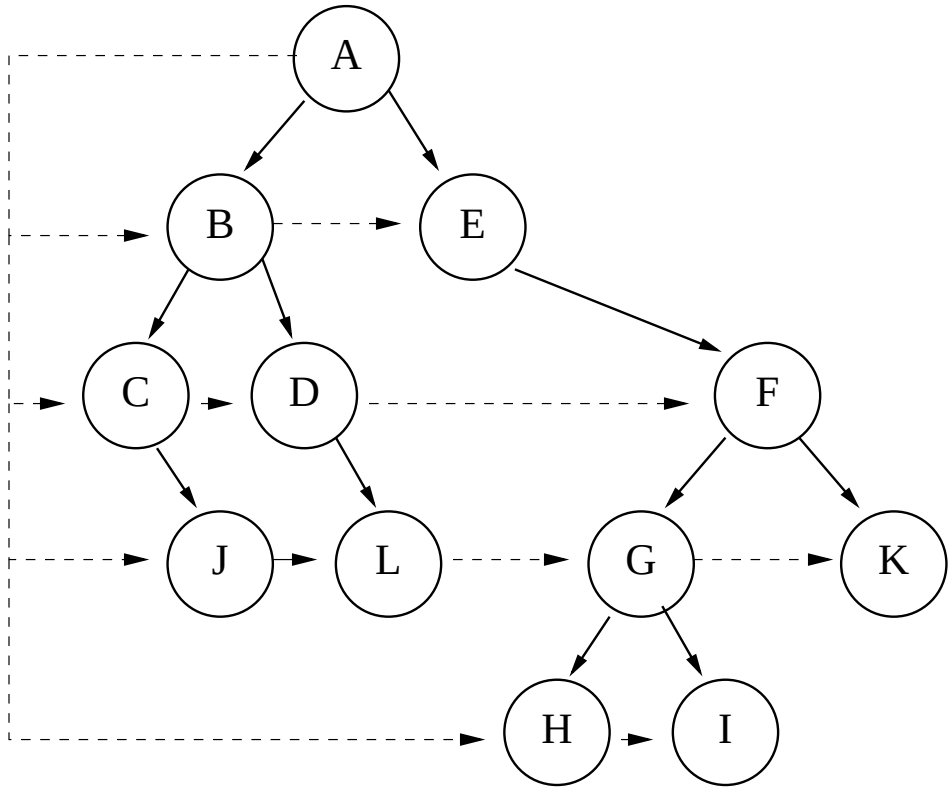
```
typedef struct Node {
   char letter;
   struct Node *left;
   struct Node *right;
}
```

Searching a Tree

This is a pre-order tree traversal:

```
Node *InTree(char l, Node *t)
{

   Node *p;

   if(t == NULL) return NULL;

   if(t->letter == l) return t;

   if((p = InTree(l, t->left)) != NULL)
      return p;

   if((p = InTree(l, t->right)) != NULL)
      return p;

   return NULL;

}
```

Level Order Traversal



To achieve this we need to use a Queue.

Queues

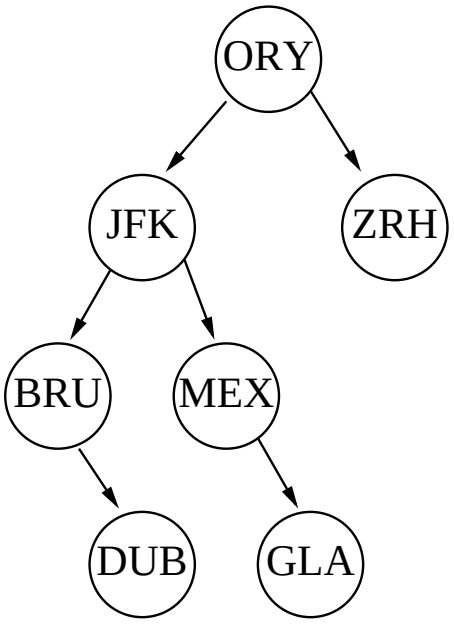
```
#define MAX_QUEUE 100
struct queue{
   Node *n[MAX_QUEUE];
   int front;
   int back;
};
typedef struct queue Queue;

void InitialiseQueue(Queue *q)
{
   q->front = 0;
   q->back = 0;
}

Node *Remove(Queue *q)
{
   Node *n;
   n = q->n[q->front];
   q->front = (q->front + 1)%MAX_QUEUE;
   return n;
}

void Insert(Node *t, Queue *q)
{
   q->n[q->back] = t;
   q->back = (q->back + 1)%MAX_QUEUE;
}

int Empty(Queue q)
{
   if(q.front == q.back) return 1;
   return 0;
}
```

Level Order Traversal

```
void PrintLevelOrder(Node *t)
{
   Queue q;
   Node *n;

   InitialiseQueue(&q);
   Insert(t, &q);
   while(!Empty(q)){
      n = Remove(&q);
      if(n != NULL){
         printf("%c\n", n->letter);
         Insert(n->left,  &q);
         Insert(n->right, &q);
      }
   }
}
```

Binary Search Trees

In a binary search tree the left-hand tree of a parent contains all keys less than the parent node, and the right-hand side all the keys greater than the parent node.



Binary Search Trees

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#define STRSIZE 500
struct node{
   int num;
   struct node *left;
   struct node *right;
};
typedef struct node Node;
Node *CreateNode();
char *PrintTree(Node *t);
void InsertBinTree(Node *t, int i);
int InTree(Node *t, int i);
```

Binary Search Trees

```
int main(void)
{

   Node *top;
   int i;
   assert(scanf("%d", &i) == 1);
   top = CreateNode();
   top->num = i;
   while(scanf("%d", &i) == 1)
      InsertBinTree(top, i);
   printf("%s\n", PrintTree(top));
   while(scanf("%d", &i) == 1)
      printf("%d is%s in Tree\n",
         i, InTree(top, i) ? "" : " not");
   return 0;

}
```

Binary Search Trees

```
void InsertBinTree(Node *t, int i)
{
   Node *p;
   if(t == NULL || i == t->num) return;
   if(i < t->num){
      if(t->left == NULL){
         p = CreateNode();
         p->num = i;
         t->left = p;
      }
      else
         InsertBinTree(t->left, i);
   }
   else{
      if(t->right == NULL){
         p = CreateNode();
         p->num = i;
         t->right = p;
      }
      else
         InsertBinTree(t->right, i);
   }
}
```

Binary Search Trees

```
char *PrintTree(Node *t)
{
   char *str;

   str = calloc(STRSIZE, sizeof(char));
   assert(str != NULL);
   if(t == NULL){
      strcpy(str, "*");
      return str;
   }
   sprintf(str, "%d(%s)(%s)", t->num,
      PrintTree(t->left),
      PrintTree(t->right));
   return str;
}

int InTree(Node *t, int i)
{
   if(t == NULL) return 0;
   if(i == t->num) return 1;

   if(i < t->num)
      return InTree(t->left, i);
   else
      return InTree(t->right, i);
}
```

Example Run

```
20
10
17
30
21
30
5
20(10(5(*)(*))(17(*)(*)))(30(21(*)(*))(*))
```

Binary Search Trees

- If the root of the tree is not well chosen, or the keys to be inserted are ordered, the tree can become a linked list !

- The tree search performs best when well balanced trees are formed - large body of literature about creating & re-balancing trees.

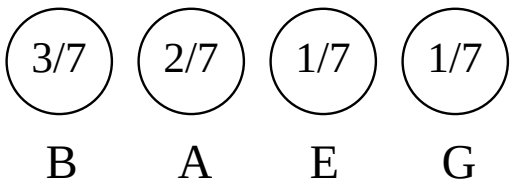
Huffman Compression

- Often we wish to compress data, to reduce storage requirements, or to speed transmission.

- Text is particularly suited to compression since using one byte per character is wasteful - some letters occur much more frequently.

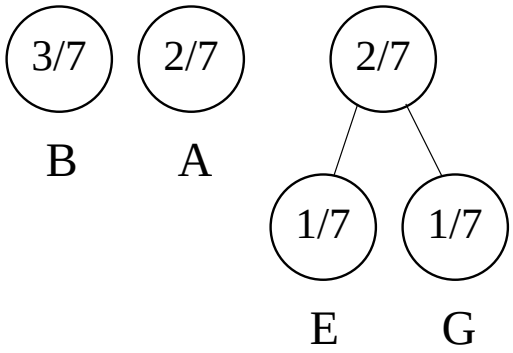
- Need to give frequently occurring letters short codes, typically a few bits. Less common letters can have long bit patterns.

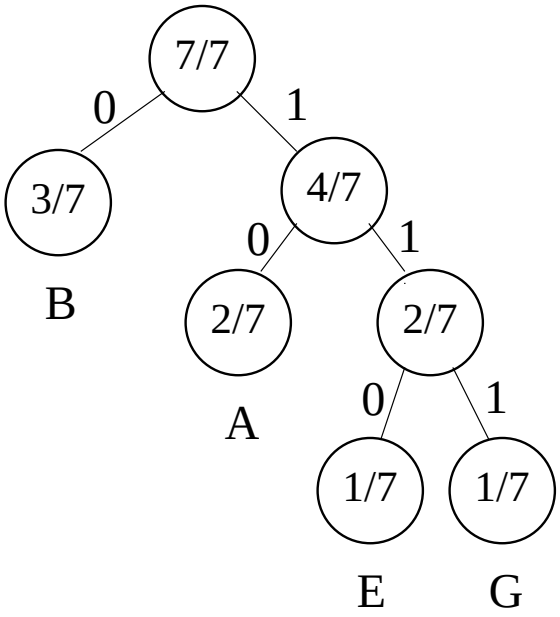
- To encode the string ”BABBAGE”



Huffman Compression

- Keep a list of characters, ordered by their frequency
- Use the two least frequent to form a sub-tree, and re-order the nodes.





- A = 10, B = 0, E = 110, G = 111
- String stored using 13 bits.

Hashing

- To keep records of employees we might index (search) them by using their National Insurance number: `xx-##-##-##-x`

- There are 17.6 billion combinations (around 2^{34}).

- Could use an array of 17.6 billion entries, which would make searching for a particular entry trivial !

- Especially wasteful since only our (5000) employees need to be stored.

- In this lecture we examine a method that, using an array of 6000 elements, would require 2.1 comparisons on average.

Hashing Nomenclature

- A hash function is a mapping, h(K), that maps from key K, onto the index of an entry.
- A black-box into which we insert a key (e.g. NI number) and out pops an array index.
- As an example lets use an array of size 11 to store some airport codes, e.g. `PHL`, `DCA`, `FRA`.
- In a three letter string $X_2X_1X_0$ the letter 'A' has the value 0, 'B' has the value 1 etc.
- One hash function is:

$$h(K) = (X_2 * 26^2 + X_1 * 26 + X_0)\%11$$

- Applying this to "DCA":

$$h(''DCA'') = (3 * 26^2 + 2 * 26 + 0)\%11$$

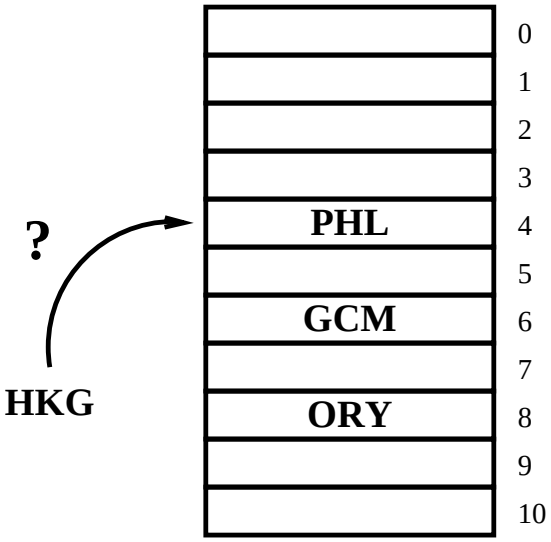
$$h(''DCA'') = (2080)\%11$$

$$h(''DCA'') = 1$$

Collisions

- Inserting ”PHL”, ”ORY” and ”GCM”:

- However, inserting ”HKG” causes a collision.



Collisions

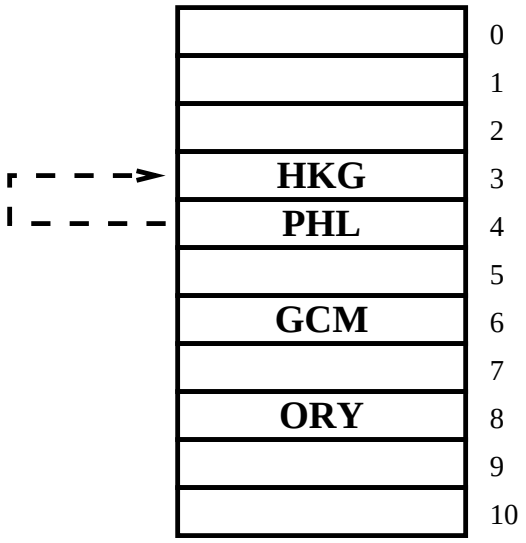
- An ideal hashing function maps keys into the array in a *uniform* and *random* manner.

- Collisions occur when a hash function maps two different keys onto the same address.

- It's very difficult to choose 'good' hashing functions.

- Collisions are common - the **von Mises** paradox. When 23 keys are randomly mapped onto 365 addresses there is a 50% chance of a collision.

Linear Probing



- The policy of finding another free location if a collision occurs is called open-addressing.
- If a collision occurs then keep stepping backwards (with wrap-around) until a free location is encountered.
- The simplest method of open-addressing is linear-probing.
- The step taken each time (probe decrement) need not be 1.
- Open-addressing through use of linear-probing is a very simple technique, double-hashing is generally much more successful.

Double Hashing

- A second function p(K) decides the size of the probe decrement.

- The function is chosen so that two keys which collide at the same address will have different probe decrements, e.g. :

  $p(K) = MAX(1, ((X_2 * 26^2 + X_1 * 26 + X_0)/11)\%11)$

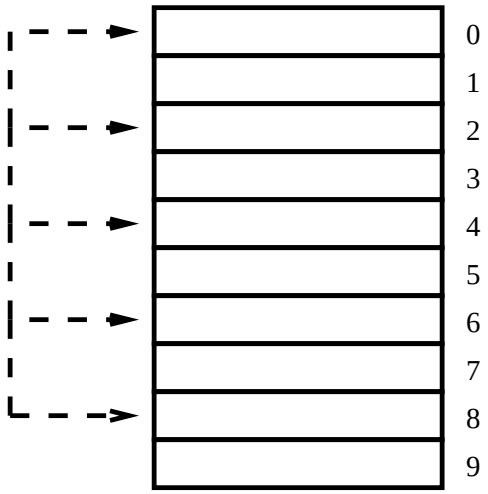
- Although ”PHL” and ”HKG” share the same primary hash value of $h(K) = 4$, they have different probe decrements:

  $p(''PHL'') = 4$

  $p(''HKG'') = 3$

Prime Array Sizes

- If the size of our array, M, was even and the probe decrement was chosen to be 2, then only half of the locations could be probed.

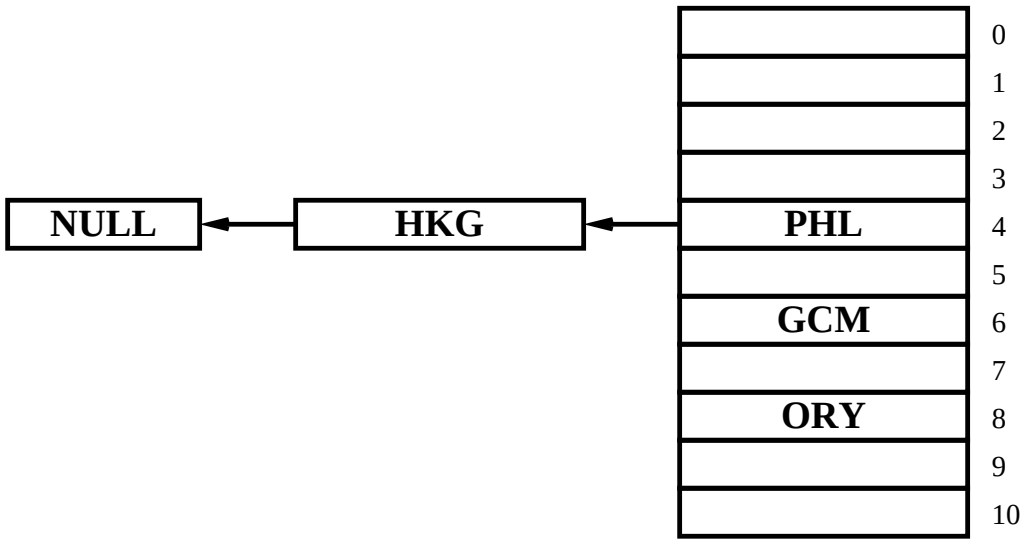


- Often we choose our table size to be a prime number and our probe decrement to be a number in the range $1 \ldots M - 1$.

- See book for other hashing methods such as folding, middle-squaring and truncation.

Separate Chaining

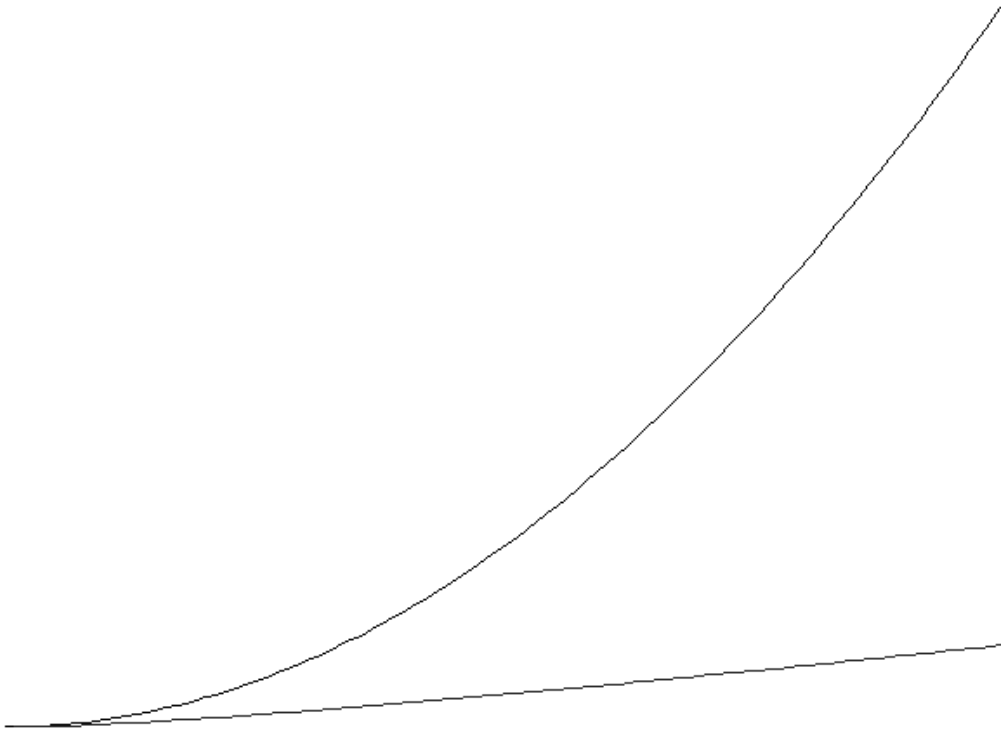
Open-addressing is not the only method of collision reduction. Another common one is separate chaining.



Sorting

- Bubblesort - we have seen this already, but at complexity $O(n^2)$ is very inefficient.

- If an algorithm uses comparison keys to decide the correct order then the theoretical lower bound on complexity is $O(n \log n)$.



Types of Sort

- Transposition (Bubblesort)
- Insertion Sort (Lab Work)
- Priority Queue (Selection sort, Heap sort)
- Divide & Conquer (Merge & Quick sort)
- Address Calculation (Proxmap)

Mergesort

The merge sort is divide-and-conquer in the sense that you divide the array into two halves, mergesort each half and then merge the two halves into order.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void mergesort(int *src, int *spare,
               int l, int r);
void merge(int *src, int *spare, int l,
           int m, int r);

#define NUM 5000

int main(void)
{

   int i;
   int a[NUM];
   int spare[NUM];

   for(i=0; i<NUM; i++)
      a[i] = rand()%100;

   mergesort(a, spare, 0, NUM-1);

   for(i=0; i<NUM; i++)
      printf("%4d => %d\n", i, a[i]);

   return 0;

}
```

Mergesort

```
void mergesort(int *src, int *spare,
               int l, int r)
{

   int m;

   if(l != r){
      m = (l+r)/2;
      mergesort(src, spare, l, m);
      mergesort(src, spare, m+1, r);
      merge(src, spare, l, m, r);
   }

}
```

Mergesort

```
void merge(int *src, int *spare,
           int l, int m, int r)
{

   int s1, s2, d;

   s1 = l;
   s2 = m+1;
   d = l;

   do{
      if(src[s1] < src[s2])
         spare[d++] = src[s1++];
      else
         spare[d++] = src[s2++];
   }while((s1 <= m) && (s2 <= r));

   if(s1 > m)
      memcpy(&spare[d], &src[s2],
             sizeof(spare[0])*(r-s2+1));
   else
      memcpy(&spare[d], &src[s1],
             sizeof(spare[0])*(m-s1+1));
   memcpy(&src[l], &spare[l],
          (r-l+1)*sizeof(spare[0]));
}
```

Quicksort

Quicksort is also divide-and-conquer. Choose some value in the array as the *pivot* key. This key is used to divide the array into two partitions. The left partition contains keys $\leq$ pivot key, the right partition contains keys $>$ pivot. Once again, the sort is then applied recursively.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int partition(int *a, int l, int r);
void quicksort(int *a, int l, int r);

#define NUM 100000

int main(void)
{

   int i;
   int a[NUM];

   for(i=0; i<NUM; i++)
      a[i] = rand()%100;

   quicksort(a, 0, NUM-1);

   return 0;

}
```

Quicksort

```
void quicksort(int *a, int l, int r)
{

   int pivpoint;

   pivpoint = partition(a, l, r);
   if(l < pivpoint)
      quicksort(a, l, pivpoint-1);
   if(r > pivpoint)
      quicksort(a, pivpoint+1, r);

}
```

Quicksort

```
int partition(int *a, int l, int r)
{

   int piv;

   piv = a[l];
   while(l<r){
      while(piv < a[r] && l<r) r--;
      if(r!=l){
         a[l] = a[r];
         l++;
      }
      /* Left -> Right Scan */
      while(piv > a[l] && l<r) l++;
      if(r!=l){
         a[r] = a[l];
         r--;
      }
   }
   a[r] = piv;
   return r;

}
```

- Theoretically both methods have a complexity $O(n \log n)$, although quicksort is preferred because it requires less memory and is generally faster.

- Quicksort can go badly wrong if the pivot key chosen is either the maximum or minimum value in the array.

Qsort()

Quicksort is so loved by programmers that a general version of it exists in ANSI C. If you need an off-the-shelf sort, and speed isn't too crucial, see `man qsort`:

```
#include <stdio.h>
#include <stdlib.h>

int intcompare(const void *a, const void *b);

int main(void)
{

   int a[10];
   int i;

   for(i=0; i<10; i++){
      a[i] = 9 - i;
   }

   qsort(a, 10, sizeof(int), intcompare);

   for (i=0; i<10; i++) printf(" %d",a[i]);
   printf("\n");
   return 0;

}
```

Qsort()

```
int intcompare(const void *a, const void *b)
{
    const int *ia = (const int *)a;
    const int *ib = (const int *)b;
    return *ia  - *ib;
}
```

Radix Sort

- The radix sort is also know as the bin sort, a name derived from its origin as a technique used on (now obsolete) card sorters.

- For integer data, repeated passes of radix sort focus on the right digit (the 1's), then the second digit (the 10's) and so on.

- Strings can be sorted in a similar manner.

Radix Sort

459 254 472 534 649 239 432 654 477

0
1
2 472 432
3
4 254 534 654
5
6
7 477
8
9 459 649 239

Read out the new list:

472 432 254 534 654 477 459 649 239

Radix Sort

472 432 254 534 654 477 459 649 239

0
1
2
3 432 534 239
4 649
5 254 654 459
6
7 472 477
8
9

432 534 239 649 254 654 459 472 477

Radix Sort

432 534 239 649 254 654 459 472 477

0
1
2 239 254
3
4 432 459 472 477
5 534
6 649 654
7
8
9

239 254 432 459 472 477 534 649 654

Radix Sort

- This has complexity $O(n)$.

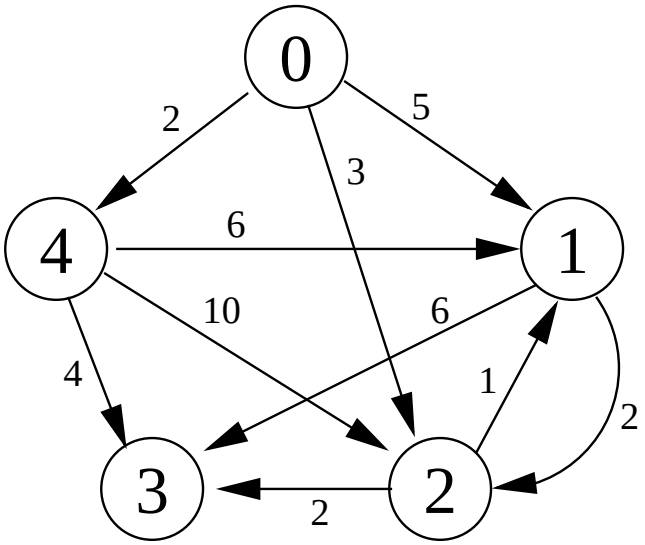
- However this simply means that the number of operations can be bounded by $k.n$, for some constant k.

- With the radix sort, k is often very large. for many lists this will be a less efficient sort than more traditional $O(n \log n)$ algorithms.

- It is difficult to write an all-purpose radix sort - you need a different one for doubles, integers, strings etc.

Graphs & Strings

- A graph, G, consists of a set of vertices (nodes), V, together with a set of edges (links), E, each of which connects two vertices.



- This is a directed graph (digraph). Vertices are joined to adjacent vertices by these edges.

- Every edge has a non-negative weight attached which may correspond to time, distance, cost etc.

Abstract Data Types

The graph type could be implemented in a large number of different ways.

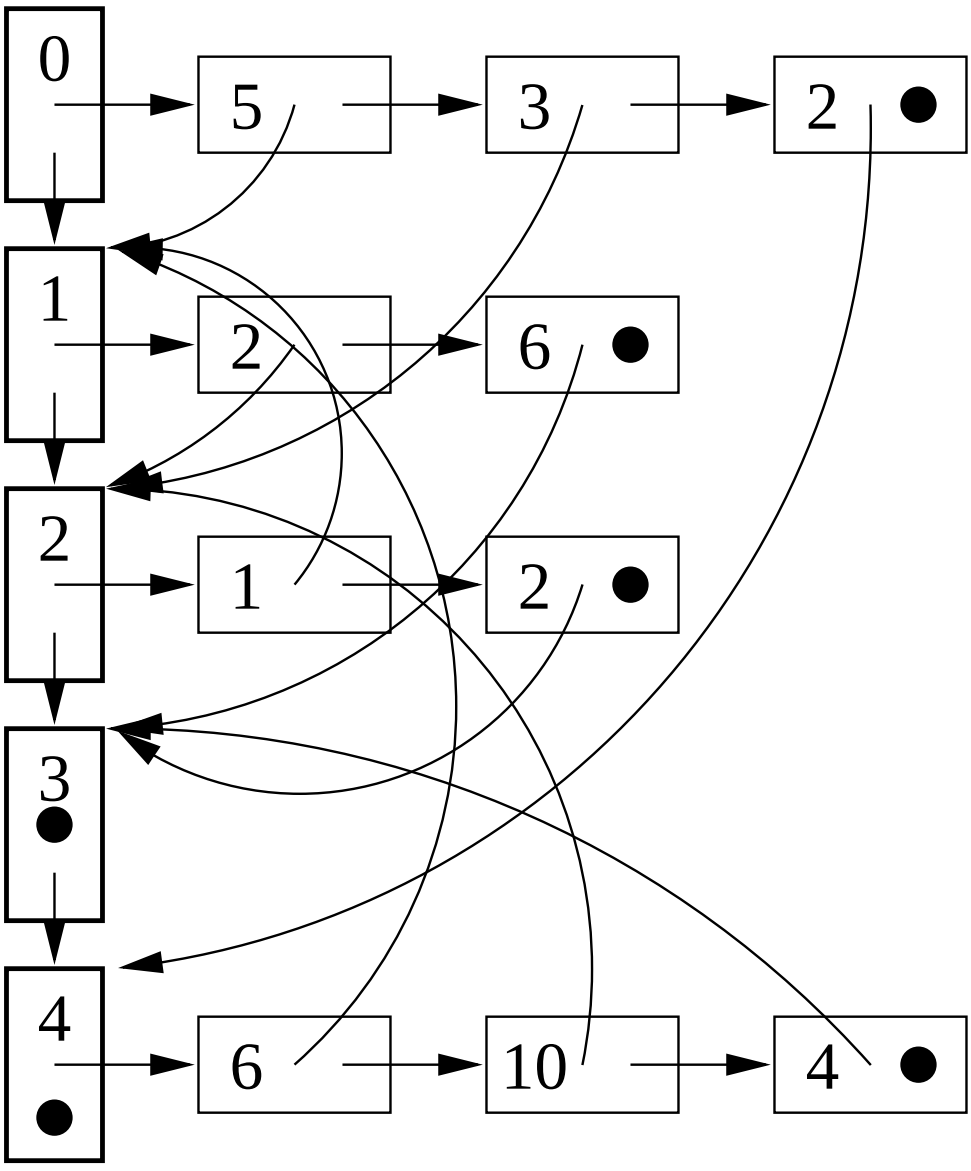
- As two sets, one for vertices, one for edges. We haven't looked at an implentation for sets, but one could use lists.

- As an adjacency table - simply encode the weighted edges in a 2D array.

	0	**1**	**2**	**3**	**4**
0	0	5	3	∞	2
1	∞	0	2	6	∞
2	∞	1	0	2	∞
3	∞	∞	∞	0	∞
4	∞	6	10	4	0

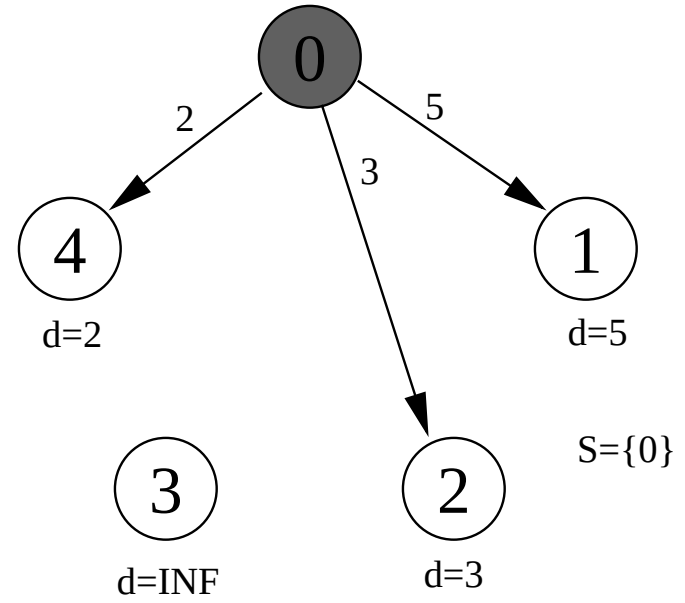
Abstract Data Types

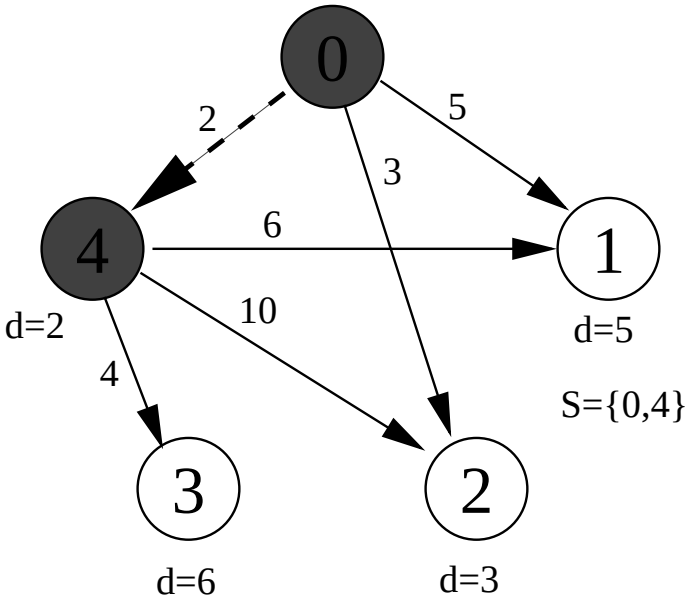
- As a Linked list:



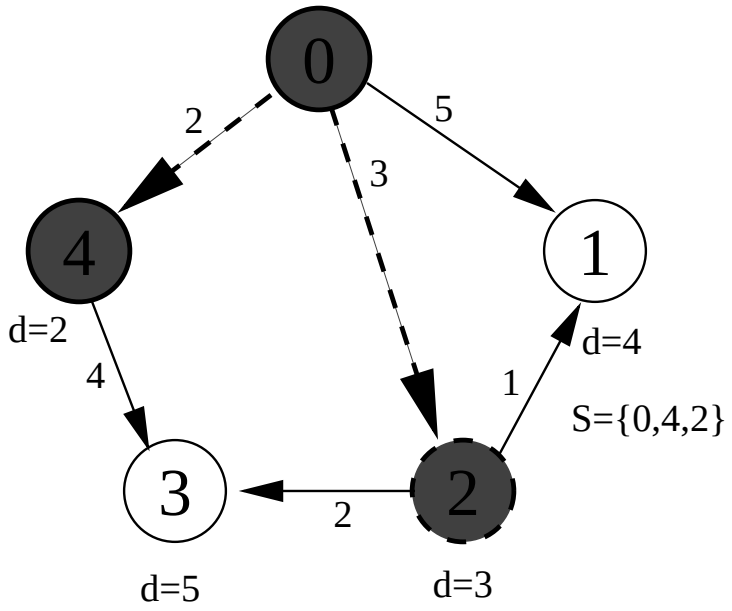
Shortest Path

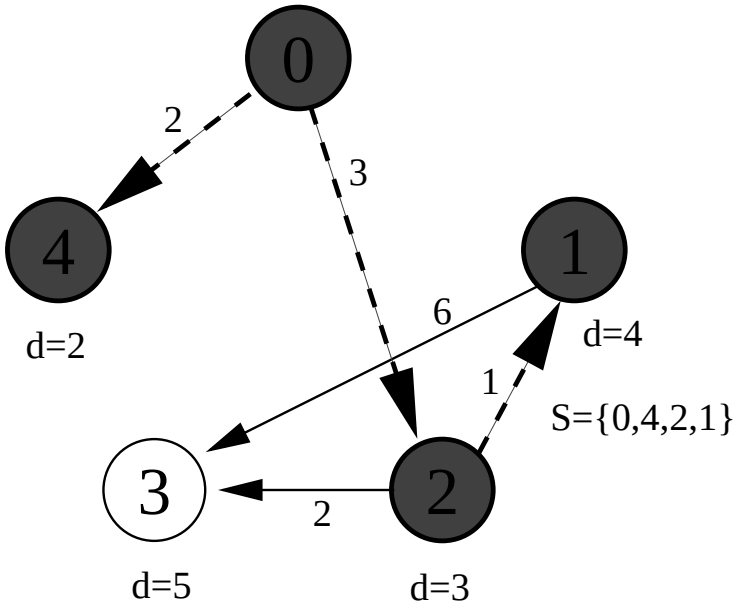
- Its often important to find the shortest path through a graph from one vertex to all others.
- One way of doing this is the greedy algorithm.



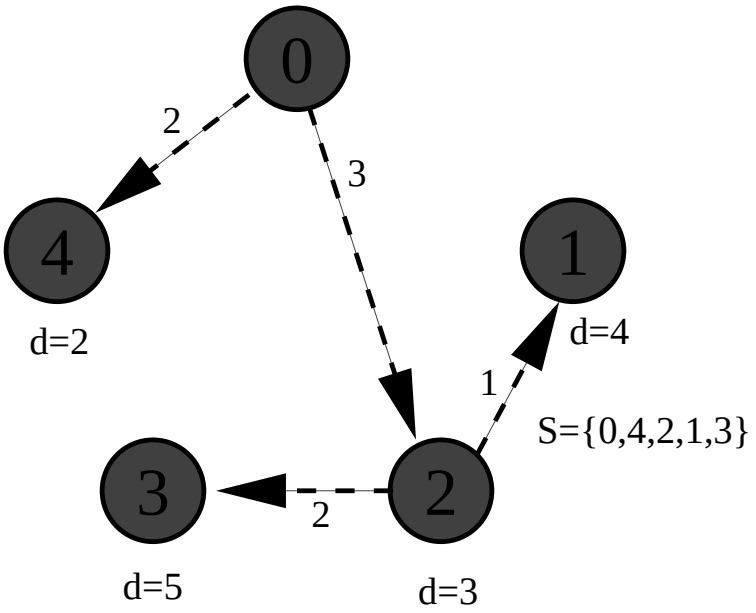


Greedy Algorithm





Greedy Algorithm



- Mark start node as distance 0.
- Mark the closest to the start.
- Check the closest to marked nodes.
- Choose closest one and edges that prove it.

Greedy Algorithm

```
#include <stdio.h>
#include <limits.h>

#define INF INT_MAX
#define MAXVERTS 50

typedef enum {FALSE, TRUE} Boolean;

typedef struct graph{
        int cost[MAXVERTS][MAXVERTS];
        int n;
} Graph;

void distance(Graph g, int d[MAXVERTS]);

int main(void)
{

   int i;
   Graph g = { {{0,  5,   3,   INF, 2  },
               {INF, 0,   2,   6,   INF},
               {INF, 1,   0,   2,   INF},
               {INF, INF, INF, 0,   INF},
               {INF, 6,   10,  4,   0 }},
               5 };
   int d[MAXVERTS];

   distance(g, d);
   for(i=0; i<g.n; i++)
      printf("0=>%d dist %d\n", i, d[i]);

   return 0;

}
```

Greedy Algorithm

```
void distance(Graph g, int d[MAXVERTS])
{

   Boolean final[MAXVERTS];
   int i, w, v, min;
   final[0] = TRUE;
   d[0] = 0;
   for(v=1; v<g.n; v++){
      final[v] = FALSE;
      d[v] = g.cost[0][v];
   }
   for(i=1; i<g.n; i++){
      min = INF;
      /* Find closest v to 0 */
      for(w=1; w<g.n; w++){
         if(!final[w]){
            if(d[w] < min){
               v = w;
               min = d[w];
            }
         }
      }
      /* Add v to S */
      final[v] = TRUE;
      /* Update remaining distances in d */
      for(w=1; w<g.n; w++){
         if(!final[w]){
            if(min + g.cost[v][w] < d[w]){
               d[w] = min + g.cost[v][w];
            }
         }
      }
   }

}
```

String Searching

- The task of searching for a string amongst a large amount of text is commonly required in word-processors for instance.

- How difficult can it be ? Don't you just do a character by character brute-force seach ?

```
Master String : AAAAAAAAAAAAH
Substring     : AAAAAAH
Substring     :  AAAAAAH
Substring     :   AAAAAAH
```

- If the master string has m characters, and the search string has n characters then this search has complexity: $O(mn)$

Rabin-Karp

Recall that to compute a hash function on a word we did something like:

$$h(''NEILL'') =$$

$$(13 \times 26^4 + 4 \times 26^3 + 8 \times 26^2 + 11 \times 26 + 11)\%P$$

where P is a big prime number. This can be expanded by Horner's method to:

$$((((((13 \times 26) + 4) \times 26) + 8) \times 26) + 11) \times 26 + 11)\%P$$

The problem here is that for a large search string, overflow can occur. We therefore move the *mod* operation inside the brackets:

$$((((((13 \times 26) + 4)\%P \times 26) + 8)\%P \times 26) + 11)\%P \times 26 + 11)\%P$$

We can compute a hash number for the search string, and for the initial part of the master string. When we compute the hash number for the next part of the master, most of the computation is common, we just need to take out the effect of the first letter and add in the effect of the new one.

One small calculation each time we move one place left in the master.

Complexity $O(m + n)$ roughly, but need to check that two identical hash numbers really has identified two identical strings.

Rabin-Karp

```
#include <stdio.h>
#include <string.h>

#define Q 33554393
#define D 26
#define index(C) (C-'A')

int rk(char *p, char *a);

int main(void)
{
   printf("%d\n", rk("LOT", "SLOT"));
   return 0;
}
int rk(char *p, char *a)
{

   int i, dM = 1, h1=0, h2=0;
   int m = strlen(p);
   int n = strlen(a);
   for(i=1; i<m; i++) dM = (D*dM)%Q;
   for(i=0; i<m; i++){
      h1 = (h1*D+index(p[i]))%Q;
      h2 = (h2*D+index(a[i]))%Q;
   }
   /* h1 = search string hash */
   /* h2 = master hash */
   for(i=0; h1!=h2; i++){
      h2 = (h2+D*Q-index(a[i])*dM) % Q;
      h2 = (h2*D+index(a[i+m])) % Q;
      if(i>n-m) return n;
   }
   return i;
}
```

Boyer-Moore

The Boyer-Moore algorithm uses (in part) an array flagging which characters form part of the search string and an array telling us how far to slide right if that character appears in the master and causes a mismatch.

```
A STRING SEARCHING EXAMPLE CONSISTING OF ...
STING    |   |    |    |    |    |
     STING   |    |    |    |    |
         STING    |    |    |    |
              STING    |    |    |
                   STING    |    |
                        STING    |
                             STING
```

- With a right-to-left walk through the search string we see that the G and the R mismatch on the first comparison. Since R doesn't appear in the search string, we can take 5 steps to the left.

- The next comparison is between the G and the S. We can slide the search string right until it matches the S in the master.

Boyer-Moore

```
A STRING SEARCHING EXAMPLE CONSISTING OF ...
STING    |   |    |    |    |    |
     STING   |    |    |    |    |
         STING    |    |    |    |
              STING    |    |    |
                   STING    |    |
                        STING    |
                             STING
```

- Now the C doesn't appear in the master and once again we can slide a full 5 places to the right.

- After 3 more full slides right we arrive at the T in CONSISTING. We align the T's, and have found our match using 7 compares (plus 5 to verify the match).

Simple Parsing

- Several fundamental algorithms have been developed to recognise legal computer programs (or expressions) and to decompose their structure into a form more suitable for processing.

- This operation, know as parsing, has application beyond Computer Science, since it is directly related to the study of the structure of languages in general.

- In mathematics we use parentheses, brackets and braces to indicate the boundaries of sub-expressions. In properly formed expressions, the various types of parentheses occur in matching pairs:

```
{a*a-[(b+c)*(b+c)-(d+e)]*[sin(x-y)]}-cos(x+y)
```

- We could use a stack to check whether or not such algebraic expressions have properly balanced parentheses.

Simple Brackets

```
#include <stdio.h>
#include <assert.h>

/* See previous stack notes */
#include "stack.h"

#define MAXEXPR 400
void CheckBrackets(char *str);
int MatchBracket(char c, char d);

int main(void)
{

   char name[MAXEXPR];
   if(!gets(name)){
      printf("Cannot read string ?\n");
      exit(2);
   }

   CheckBrackets(name);

   return 0;

}

int MatchBracket(char c, char d)
{

   if(c == '{' && d == '}') return 1;
   if(c == '(' && d == ')') return 1;
   if(c == '[' && d == ']') return 1;

   return 0;

}
```

Simple Brackets

```
void CheckBrackets(char *str)
{

   char c;
   Stack s;
   InitialiseStack(&s);

   while(*str){

      if(*str == '{' || *str == '(' ||
         *str == '['){
         Push(&s, (int)*str);
      }
      if(*str == '}' || *str == ')' ||
         *str == ']'){
         c = Pop(&s);
         if(!MatchBracket(c, *str)){
            printf("Parse Error !\n");
            exit(2);
         }
      }
      str++;
   }

   if(!Empty(&s)){
      printf("Parse Error !\n");
      exit(2);
   }
   printf("Everything OK\n");

}
```

Reverse Polish

- It is a long standing tradition in mathematics to write the operator between the operands, as in `x+y`, rather than `x y +`.

- The 'normal' method with the operator between the operands is known as *infix* notation.

- The alternative is called *Postfix* or *Reverse Polish* after the Polish logician J. Lukasiewicz (1958) who investigated the properties of this notation.

- As you scan a traditional infix expression, such as:

  ```
  A+B/C+D
  ```

  from left- to right, it is impossible to tell when you initially encounter the `+` sign whether or not you should apply the indicated operation to A.

- You must probe deeper into the expression to determine whether an operation with a higher priority occurs.

- This gets very complicated.

- Brackets make this even worse.

Reverse Polish Notation

Infix:

```
A+B*C
A*B+C
((A+B)*C+D)/(E+F+G)
```

Reverse Polish

```
ABC*+
AB*C+
AB+C*D+EF+G+/
```

Notice that no brackets are required in Reverse Polish. Therefore, for simple applications, we could require the user to enter expressions in postfix form.

Implementing Reverse Polish

- Reverse Polish may be evaluated by use of a stack.
- Examine the next character;
- If it is a number (or variable in the general case) push it onto the stack.
- If it is an operator (+-/*), pop off the top two items, perform the operation and push the result.
- If we have reached the end the answer is the one and only item on the stack. Else repeat.

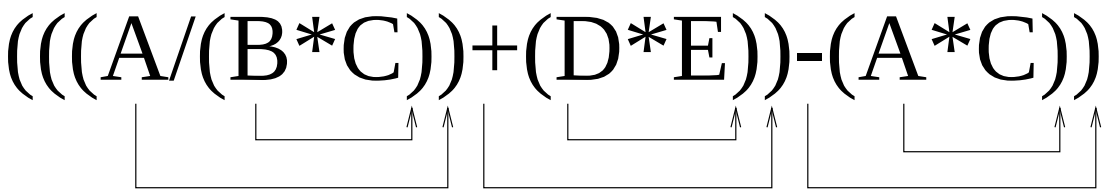
```
8 2 5 * + 1 3 2 * + 4 - /
```

	*	+		*	+	-		/
5 2 8	10 8	18	2 3 1 18	6 1 18	7 18	4 7 18	3 18	6

Completely Parenthesised Expressions

- How do we convert from infix to (the more easily handled) postfix ?

- If your expression is comletely parenthesised:
  - Move each operator to the space held by its corresponding right (closing) parenthesis.
  - Remove all parentheses.



Completely Parenthesised Expressions

```
#include <stdio.h>
#include <string.h>

#define MAXSTR 400

void Move(char *q, int b, char *spare);

int main(void)
{

   int brk = 0;
   char str[]="((((A+B)*C)+D)/((E+F)+G))";
   char *p = str;
   char *s;
   char *k;

   s = (char *)strdup(str);
   k = s;
```

Completely Parenthesised Expressions

```
   while(*p){
      if(*p == '(') brk++;
      if(*p == ')') brk--;
      if(*p == '*' || *p == '/' ||
         *p == '-' || *p == '+'){
         Move(p, brk, s);
         *s = ' ';
      }
      p++;
      s++;
   }
   s = k;
   while(*s){
      if(*s != '(' && *s != ' ')
         putchar(*s);
      s++;
   };
   printf("\n");

   return 0;
}
```

Completely Parenthesised Expressions

```
void Move(char *q, int b, char *spare)
{

   char o;
   int brk = b-1;

   o = *q;
   do{
      q++;
      spare++;
      if(*q == '(') b++;
      if(*q == ')') b--;
   }while(*q != ')' || b != brk);
   *spare = o;

}
```

General Infix to Postfix

- Completely parenthesised expressions are very rare, and not very general.

- A totally general approach for translating infix to postfix relies on a table of operator precedence:

*	/	+	-	(	)	'\0'
2	2	1	1	3	0	0

General Infix to Postfix

```
#include <stdio.h>
#include <ctype.h>
#include <assert.h>
#include "stack.h"
#include "queue.h"

void PrintPostFix(char *infix);
int isoperator(char c);

int main(void)
{

   char infx[]="(((A/(B*C))+(D*E))-(A*C))";
   PrintPostFix(infx);
   return 0;

}

int isoperator(char c)
{
   if(c == '+' || c == '-' ||
      c == '*' || c == '/'){
      return 1;
   }
   else{
      return 0;
   }
}
```

General Infix to Postfix

```
void PrintPostFix(char *infix)
{

   Queue q;
   Stack s;
   int ip = 0;
   char c, d;
   int priority[256];

   InitialiseQueue(&q);
   InitialiseStack(&s);
   Push(&s, '\0');

   priority['*'] = 2; priority['/'] = 2;
   priority['+'] = 1; priority['-'] = 1;
   priority['('] = 3; priority[')'] = 0;

   do{
      c = infix[ip++];
      if(isupper(c))
         InsertQueue(c, &q);
      else if(c == ')'){
            d = Pop(&s);
            while(d != '('){
               InsertQueue(d, &q);
               d = Pop(&s);
            }
      }
      else if(c == '\0'){
               while(!EmptyStack(&s)){
                  d = Pop(&s);
                  InsertQueue(d, &q);
               }
      }
```

General Infix to Postfix

```
      else if(c == '(' || isoperator(c)){
         d = Pop(&s);
         while(priority[(int)d] >= priority[(int)c]
               && isoperator(d)){
            InsertQueue(d, &q);
            d = Pop(&s);
         }
         Push(&s, d);
         Push(&s, c);
      }
      else{
         printf("Invalid Token \"%c\"\n", c);
         exit(2);
      }
   }while(c != '\0');

   for(ip=0; !EmptyQueue(q); ip++)
      printf("%c", RemoveQueue(&q));
   printf("\n");

}
```

Formal Grammars

- Parsing a program is the process of grammatically analysing how it is composed into parts.

- Before we can write a program to determine whether a program written in a given language is legal, we need a description of exactly what constitutes a legal program.

- Programming languages are often described by a particular type of grammar called a *context free grammar*.

- Say we wish to create a new computer language whose sole purpose is to print out noughts and ones onto the screen :

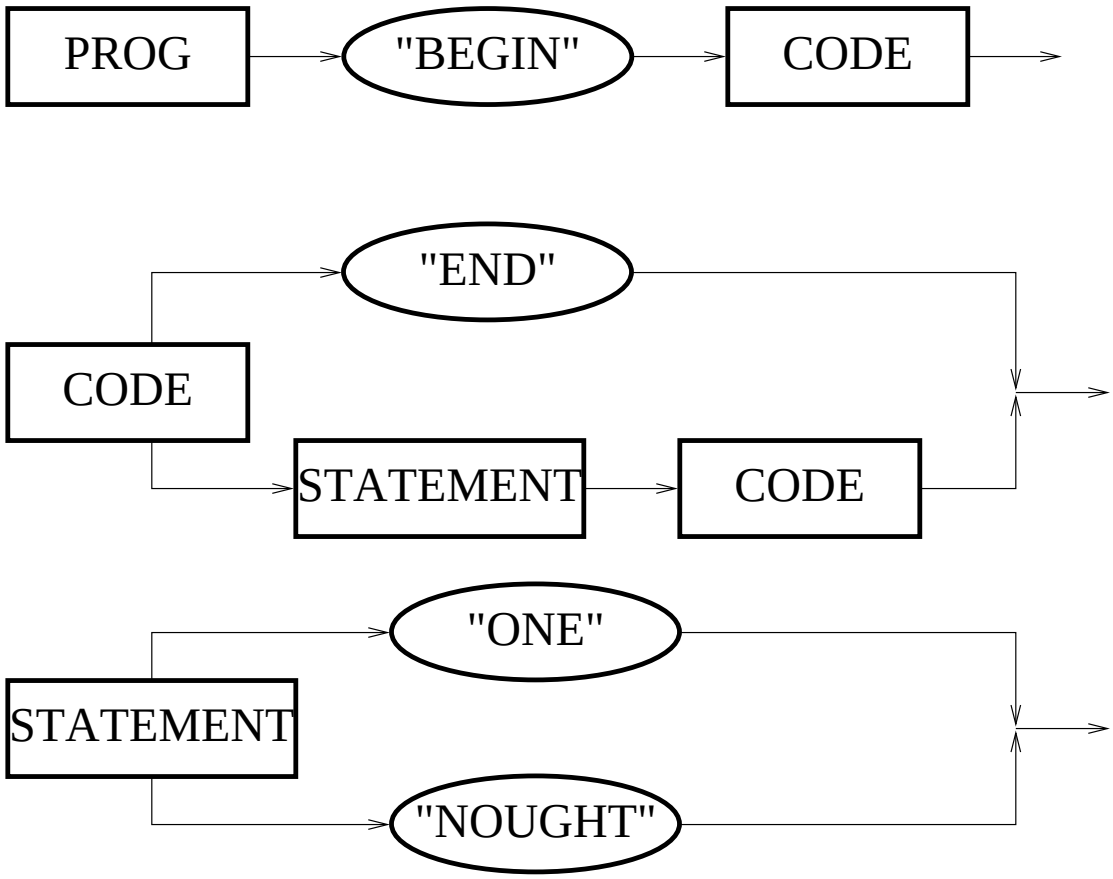
```
BEGIN
     ONE
     NOUGHT
     ONE
END
```

- We will need a formal definition of the language.

0's & 1's Example

```
<PROG>      ::= "BEGIN" <CODE>
<CODE>      ::= "END" | <STATEMENT> <CODE>
<STATEMENT> ::= "ONE" | "NOUGHT"
```

- The '|' means OR.
- "BEGIN", "ONE" and "NOUGHT" are string constants.
- <CODE> is described recursively.
- You could also think of this grammar in terms of a *railroad diagram*:



0's & 1's Example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

#define MAXNUMTOKENS 100
#define MAXTOKENSIZE 7
#define PROGNAME "01.no"
#define strsame(A,B) (strcmp(A, B)==0)
#define ERROR(PHRASE) {fprintf(stderr,
"Fatal Error %s occured in %s, line %d\n",
PHRASE, __FILE__, __LINE__); exit(2); }

struct prog{
   char wds[MAXNUMTOKENS][MAXTOKENSIZE];
   int cw; /* Current Word */
};
typedef struct prog Program;

void Prog(Program *p);
void Code(Program *p);
void Statement(Program *p);
```

0's & 1's Example

```
int main(void)
{
   int i;
   FILE *fp;
   Program prog;

   prog.cw = 0;
   for(i=0; i<MAXNUMTOKENS; i++)
      prog.wds[i][0] = '\0';
   if(!(fp = fopen(PROGNAME, "r"))){
      fprintf(stderr, "Cannot open %s\n",
               PROGNAME);
      exit(2);
   }
   i=0;
   while(fscanf(fp, "%s", prog.wds[i++])==1
         && i<MAXNUMTOKENS);
   assert(i<MAXNUMTOKENS);
   Prog(&prog);
   printf("Parsed OK\n");
   return 0;
}
```

0's & 1's Example

```
void Prog(Program *p)
{
   if(!strsame(p->wds[p->cw], "BEGIN"))
      ERROR("No BEGIN statement ?");
   p->cw = p->cw + 1;
   Code(p);
}

void Code(Program *p)
{
   if(strsame(p->wds[p->cw], "END"))
      return;
   Statement(p);
   p->cw = p->cw + 1;
   Code(p);
}

void Statement(Program *p)
{
   if(strsame(p->wds[p->cw], "ONE")){
      return;
   }
   if(strsame(p->wds[p->cw], "NOUGHT")){
      return;
   }
   ERROR("Expecting a ONE or NOUGHT ?");
}
```

0’s & 1’s Example

```
BEGIN
   ONE
   NOUGHT
   ONE
END
```
Parsed OK

```
BEGIN ONE NOUGHT NOUGHT END
```
Parsed OK

```
BEGIN END
```
Parsed OK

```
BEGIN
  ONE
  TWO
END
```
Fatal Error Expecting a ONE or NOUGHT ? occured in p01a.c, line 79

```
BEGIN
  ONE
  NOUGHT
```
Fatal Error Expecting a ONE or NOUGHT ? occured in p01a.c, line 79

```
  ONE
  NOUGHT
END
```
Fatal Error No BEGIN statement ? occured in p01a.c, line 55

0's & 1's Example

- Notice that the END statement is actually used as the recursive base-case in the formal grammar in the function Code().

- Notice that the parser doesn't actually do anything other than check that the input has the correct syntax.

- An interpreter both checks the syntax and performs the required operations.

- A slight modification to the code is required to produce an interpreter :

```
void Statement(Program *p)
{
   if(strsame(p->wds[p->cw], "ONE")){
      printf("1\n");
      return;
   }
   if(strsame(p->wds[p->cw], "NOUGHT")){
      printf("0\n");
      return;
   }
   ERROR("Expecting a ONE or NOUGHT ?");
}
```

Mathematical Expressions

To parse a string such as ”A+B*C”, ”A*(B+C)” or ”-(B*F)” we use :

```
<EXPR> ::= <EXPR><OP><EXPR> |
           "(" <EXPR> ")" |
           "-"<EXPR> | Letter
<OP>   ::= "+" | "-" | "*" | "/"
```

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define MAXEXPR 400

struct prog{
        char str[MAXEXPR];
        int count;
};
typedef struct prog Prog;

void Op(Prog *p);
int isop(char c);
void Expr(Prog *p);
```

Mathematical Expressions

```
int main(void)
{

   Prog p;
   p.count = 0;

   if(scanf("%[A-Z,+,-,*,/,(,)]s", p.str) != 1){
      printf("Couldn't read your expression ?\n");
      exit(2);
   }
   Expr(&p);

   printf("Parsed OK !\n");
   return 0;

}

int isop(char c)
{

   if(c=='+' || c=='-' || c=='*' || c=='/')
      return 1;
   else
      return 0;
}

void Op(Prog *p)
{
   if(!isop(p->str[p->count])){
      printf("I was expecting a letter ?\n");
      exit(2);
   }
}
```

```
void Expr(Prog *p)
{
   if(p->str[p->count] == '('){
      p->count = p->count + 1;
      Expr(p);
      p->count = p->count + 1;
      if(p->str[p->count] != ')'){
         printf("I was expecting a ) ?\n");
         exit(2);
      }
   }

   else if(p->str[p->count] == '-'){
      p->count = p->count + 1;
      Expr(p);
   }

   /* Note Look-Ahead */
   else if(isop(p->str[p->count+1])){
      if(isupper(p->str[p->count])){
         p->count = p->count + 1;
         Op(p);
         p->count = p->count + 1;
         Expr(p);
      }
   }
   else{
      if(!isupper(p->str[p->count]) ||
         isupper(p->str[p->count+1])){
         printf("Expected a single letter ?\n");
         exit(2);
      }
   }

}
```

Mathematical Expressions

```
A+(B*C)
```

Parsed OK !

```
-(B*C+D)
```

Parsed OK !

```
A
```

Parsed OK !

```
A+(C*
```

I was expecting a single letter ?

```
a+c
```

Couldn’t read your expression ?

```
A*B+(C*D
```

I was expecting a) ?

Mathematical Expressions

- The formal grammar doesn't explain everything that the programmer needs to know.

- It is not clear whether the `a+c` example is invalid or not.

- It is not clear how spaces should be dealt with.