Introduction to Data Structures

Kurt Schmidt

Dept. of Computer Science, Drexel University

May 3, 2017

Intro

Objectives:

- Review the fundamental algorithms and data structures that are commonly used in programs
- To see how to use and implement these algorithms and data structures in different languages and to see what language and library support exists for them

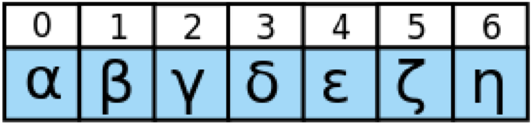
Topics

- Arrays and Vectors
- Lists
- Linear/Binary Search
- Quicksort
- Dictionaries

Vectors

Vectors (Arrays)

- Sequence of items
- Indexable
  - Same time to access any element
- (Conceptually) contiguous chunks of memory
- In CS, *array* and *vector* are interchangeable enough



Time, Operations on Vectors

- Access: constant time ($\Theta(1)$)
- Searching:
  - Sorted array – $\Theta(\log n)$
  - Unsorted – $\Theta(n)$
- Inserting, removing items:
  - Unordered – $\Theta(1)$
    - Add to end
    - Replace deleted item w/last guy
  - Ordered – $\Theta(n)$
    - Need to make (or fill in) a hole
    - Move $n/2$ items, on average, to maintain relative order

Resizing Arrays

- Many languages have arrays which manage themselves
  - Awk, Python[1], Perl[1], etc.
- Other languages have smart arrays in their library:

  C++ `vector` in the STL

  Java `ArrayList`
- This doesn't mean the operations are free
  - What goes on underneath the hood may be important
- We shall create our own machinery in C

[1] Called *lists*

Some C Memory Management Functions

`void* malloc(int n)` allocates n contiguous bytes from heap, returns address of first byte (`NULL` upon failure)

`void free(void *p)` returns to the heap memory addressed by `p`. Does *nothing* to `p` itself

`void* memmove(void* d, void* s, size_t n)` moves n bytes from `s` to (possibly overlapping) region starting at `d`

`void* memcpy(void* d, void* s, size_t n)` copies n bytes from `s` to (non-overlapping) region starting at `d`

`int sizeof()` actually an operator, returns size, in bytes, of given object or type

`void* realloc(void* src, int n)` attempts to resize array in place, or a bigger section elsewhere, copies contents for you. Returns pointer to

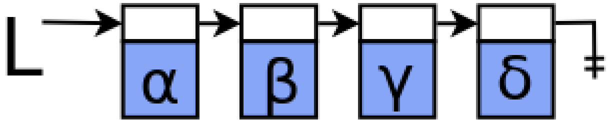
Growing Arrays in C

```
enum { INIT_SIZE=1, GROW_FACTOR=2 } ;
int curr_size = INIT_SIZE ;
int nr_elems = 0; /* # of useful elements */
int *a = (int*)malloc( INIT_SIZE * sizeof( int )) ;
... /* some stuff here */
/* attempt to insert 24 */
if( nr_elems >= curr_size ) { /* need to grow */
   int *t = realloc( a, curr_size*GROW_FACTOR*sizeof( int ))
   if( t != NULL ) { /* success! */
      curr_size *= GROW_FACTOR ;
      a = t ;
      a[nr_elems++] = 24 ;
   }
   else
      /* FAILURE! */
}
```

Lists

Lists

- A sequence of elements
- *Not* indexable (immediately)
  - To access 5^{th} element, must visit the preceding 4
- Space is allocated for each new element
- Consecutive elements are linked together with a pointer
- Middle can be modified in constant time



Lists as Ordered Pairs

- For languages w/out explicit pointers, such as Bash, Maple, Python, and Java, it might be helpful to consider a list as an ordered pair
  1. The item (payload)
  2. The rest of the list

$$(\alpha, (\beta, (\gamma, (\delta, ()))))$$

  - Where () is the empty list
- We might use a class
- Or, simply, nested arrays, of size 2 (or, empty)
  - This is a very LISP notion

Time, Operations on Lists

- Access (same as searching) – linear time ($\Theta(n)$)
- Modifying anywhere – constant time ($\Theta(1)$)
- Inserting
  - At front – $\Theta(1)$
  - Append – $\Theta(n)$, unless pointer to last element kept

Lists in Python

- Python's `list` is really an array
  - Really? How might we tell?
  - In any case, elements are accessed in constant time
- We'll use Python's `list` to hold our duples
  - We'll call them *cells*, or *nodes*
- Let the empty `list`, `[]`, be an empty list
- Remember, *everything* in Python is a reference (pointer)

“Linked” Lists in Python


```
L = []
    # add 24 to front
L = [ 24, L ]
print L
    # add 3 to the front
L = [ 3, L ]
print L
```

Would output:

```
[ 24, [] ]
[ 3, [ 24, [] ]]
```

Append to end of List

```
def append( L, e ) :
   '''Append item e to end of L
   Note, reference L doesn't change'''

   t = L  # start at beginning

   while t != [] :
      t = t[1] # move to next cell

   # We have our hands on the last cell (empty list)

   # Make it a pair, w/a new end-of-list
   t.extend( [ e, [] ] )
```

Searching a List in Python


```
def search( L, t ) :
   '''Return cell of L that contains t,
   None if not found'''

   while L != [] :
      if L[0] == t :
         return L
      L = L[1] # move to next cell

   return None # didn't find it
```

Map – Apply Function to a List

```
def apply( L, fn ) :

   while L != [] :
      fn( L )
      L = L[1] # move to next cell
```

`fn` is any function that takes a single cell, modifies it. E.g.:

```
def square( c ) :

   c[0] *= c[0]
```

Examples of Apply

Given:

```
L = [ 1, [ 2, [ 3, [] ]]]
```

Print the list:

```
def printCell( cell ) :
    print cell[0]

apply( L, printCell )
```

```
1 2 3
```

```
apply( L, square )
apply( L, printCell )
```

```
1 4 9
```

Lists in C

```
typedef struct sNode sNode ;

struct sNode { /* a node (cell) in a singly-link list */
   int data; /* the payload */
   sNode* next ;
} ;

/* Wrap an item in a node (cell) */
sNode* newNode( int d ) {
   sNode *newp ;
   newp = (sNode*) malloc( sizeof( sNode )) ;
   if( newp != NULL ) {
      newp->data = d ;
      newp->next = NULL ;
   }
   return newp ;
}

typedef sNode* List ;
```

Insert at Front of C List


```c
   /* addfront: add newp to front of listp *
    *   return ptr to new list             */
sNode* addfront( sNode *listp, sNode *newp )
{
   newp->next=listp ;
   return newp ;
}
list = addfront( list, newNode( 13 )) ;
list = addfront( list, newNode( 12 )) ;
list = addfront( list, newNode( 5 )) ;
```

List would be (5 12 13)

Append to End of C List


```c
/* append: add newp to end of listp *
 *   return ptr to new list      */
sNode* append( sNode* listp, sNode* newp )
{
   sNode *p ;
   if( listp == NULL )
      return newp ;
   for( p=listp; p->next!=NULL; p=p->next )
      ;   /* Find last node */
   p->next = newp ;
   return listp ;
}

list = append( list, newNode( 42 )) ;
```

List would be (5 12 13 42)

Search a List in C

```c
/* lookup: linear search for t in listp   *
 *   return ptr to node containing t, or NULL */
sNode* lookup( sNode *listp, int t )
{
   for( ; listp != NULL; listp = listp->next )
      if( listp->data == t )
         return listp ;

   return NULL; /* no match */
}
```

Map/Apply on List in C

```c
/* apply: execute fn for each element of listp */
void apply( sNode *listp,
            void (*fn)(sNode*, void* ), void* arg )
{
   for ( ; listp != NULL; listp = listp->next)
      (*fn)( listp, arg ); /* call the function */
}
```

- The 2nd argument is a function pointer
  - `void` return type
  - It takes 2 arguments
    1. List
    2. Generic pointer, to be used by function, as needed

Use Map to Square Elements in List

```
void squareVal( sNode *p, void *arg )
{
  /* note, arg is unused */

  p->data *= p->data ;
}

apply( list, squareVal, NULL ) ;
```

Use Map to Print

```
/* printVal: print value, using arg as format string */

void printVal( sNode *p, void *arg )
{
   char* fmt = (char*) arg ;
   printf( fmt, p->data ) ;
}

apply( list, prntVal, "%d" ) ;
```

Use Map to Compute Size

```
   /* incCounter: increment counter in arg */
void incCounter( sNode *p, void *arg )
{
   /* NOTE: p is unused. We were called, there's a node. */

   int* ip = (int*) arg ;
   (*ip)++ ;
}

int size = 0 ;
apply( list, incCounter, &size ) ;
printf( "%d elements in list\n", size ) ;
```

Freeing Nodes in a List

```c
/* freeall: free all elements of listp */
void freeall( sNode *listp )
{
   sNode *t ;
   for ( ; listp != NULL; listp = t ) {
      t = listp->next ;
      free(listp) ;
   }
}
```

What's the problem with the following?

```c
for ( ; listp != NULL; listp = listp->next)
   free( listp ) ;
```

Removing Element from List


```c
   /* delitem: delete first t from listp */
sNode *delitem( sNode *listp, int t ) {
   sNode *p, *prev = NULL ;
   for( p=listp; p!=NULL; p=p->next ) {
      if( p->data == t ) {
         if( prev == NULL ) /* front of list */
            listp = p->next ;
         else
            prev->next = p->next ;
         free( p ) ;
         break ;
      }
      prev = p ;
   }
   return listp ;
}
```

Searching & Sorting

Linear Search

- Exhaustively examine each element
- Examine each element, until you find what you seek, or you've examined every element
  - Note that order of examination doesn't matter
- The *only* search for a linked-list
- Need $\Theta(n)$ comparisons, worst and average

Linear Search on Array in C


```
   /* return index of first find, -1 otherwise */
int linSearch( int *a, int size, int t )
{
   int i ;
   for( i=0; i<size; ++i )
      if( a[i] == t )
         return i ;

   return i ;
}

int test[ 12 ] = { ... } ;
int l = linSearch( test, 12, 17 ) ;
```

Binary Search

- Only works on *sorted* collections
- Only efficient on collections with random (direct) access (vectors)
  - Find it?
- Start in the middle:
  - Find it?
  - Less than? Look in lower $\frac{1}{2}$
  - Greater than? Look in upper $\frac{1}{2}$
- Cut search space in $\frac{1}{2}$
- Need $\Theta(\log n)$ time, worst and avg.

Binary Search in C


```
   /* Search integer array
       Return index of target, or -1 */
int binSearch( int* arr, int size, int target )
{
   int low = 0,
       high = size-1 ;
   int mid ;

   while( low <= high )
   {
      mid = (low+high) / 2 ;
      if( arr[mid] == target )
         return mid ;
      if( target < arr[mid] )
         high = mid-1 ;
      else
         low = mid+1 ;
   }

   return( -1 ) ;
}
```

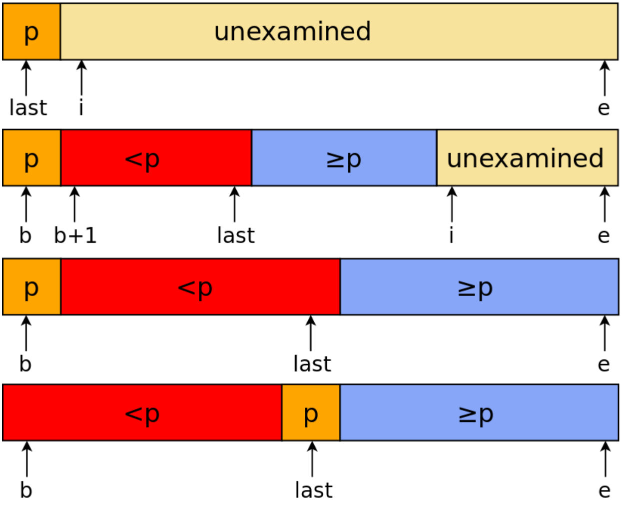
Quick Sort

- Choose one element of the array (the *pivot*)
- Partition the other elements into two groups:
  - those less than the pivot
  - those greater than or equal to the pivot
- Pivot is now in the right place
- Recursively sort each (strictly smaller) group
- Can be done in place

Quick Sort – Run Time

- Each partition requires $\Theta(n)$ comparisons, moves
- Best case – $\Theta(n \log n)$
  - Each partition splits collection in half
  - Can do that about n times
- Worst case – $\Theta(n^2)$
  - Each partition gets pivot in place
  - Leaves $n-1$ elements in one partition to sort
  - Looks like a Selection Sort
- On random data, average run time is $\Theta(n \log n)$

Quick Sort – Description



Recursive Quicksort in C

```
   /* quicksort: sort v[0]..v[n-1] into increasing order */
void quicksort( int v [], int n )
{
   if( n <= 1 )     /* nothing to do */
      return ;

   int piv = partition( v, n ) ;

   quicksort( v, piv ) ;   /* recursively sort each part. */
   quicksort( v+piv+1, n-piv-1 ) ;
}
```

Quicksort – Partition

```
   /* partition, return index of pivot */
int partition( int *v, int n )
{
   int i, last=0 ;
   swap( v,0,rand() % n ); /* move pivot element to v[0] */

   for ( i = 1; i < n; i++ )  /* partition */
      if ( v[i] < v[0] )
         swap( v, ++last, i ) ;
   swap( v, 0, last );     /* restore pivot */

   return last ;
}
```

Library Sorts for Some Languages

C	`qsort` (in `stdlib.h`)
C++	STL `sort` (in `algorithm`)
Java	`java.util.Collections.sort`)
Perl	`sort`
Python	`list.sort`, `sorted`

qsort – C Standard Library


```
qsort( void* a, int n, int s,
        int (*cmp)(void *a, void *b) ) ;
```

- Sorts the first n elements of array a
- Each element is s bytes
- cmp is a function you must provide
  - Compares 2 single elements, *a and *b
    - qsort must pass void pointers, since it doesn't know the type
    - cmp does, since you provide it
  - Returns integer -1 if a<b, 0 if a==b, and 1 if a>b

qsort Example for Integers

```
   /* icmp: integer compare of *p1 and *p2 */
int icmp( void *p1, void *p2 )
{
   int v1 = *((int*) p1) ;
   int v2 = *((int*) p2) ;
   if( v1 < v2 )
      return -1 ;
   else if( v1 == v2 )
      return 0 ;
   else
      return 1 ;
}

int arr[N] ;
...
qsort( arr, N, sizeof(arr[0]), icmp ) ;
```

qsort Example for Strings

```
   /* scmp: string compare of *p1 and *p2. p1 is a ptr to a
    * string, ptr to a char*, so is a ptr to a ptr, or a char**
    */
int scmp( void *p1, void *p2 )
{
   char *v1, *v2 ;

   v1 = *((char**) p1) ;
   v2 = *((char**) p2) ;

   return strcmp( v1, v2 ) ;
}

char *str[N] ;
...
qsort(str, N, sizeof(str[0]), scmp) ;
```

Dictionary

Dictionary (Map)

- A *set* of (key, value) pairs
- Allows us to associate satellite data w/a key
- E.g., phone book (sorta), student record, given an ID, an error string (given an error number)
- Keys are unique
- Operations:
  - Lookup (find)
  - Insert
  - Remove

Times – Simple Dictionaries

Unordered Vector

- Lookup – $\Theta(n)$
- Insertion – $\Theta(1)$ (given a find)
- Removal – $\Theta(1)$ (given a find)

Ordered Vector

- Lookup – $\Theta(\log n)$
- Insertion – $\Theta(n)$ (given a find)
- Removal – $\Theta(n)$ (given a find)

Some Other Dictionaries

Binary Search Tree[1]

- Lookup – $\Theta(\log n)$
- Insertion – $\Theta(\log n)$
- Removal – $\Theta(\log n)$

Hash Table

- Lookup – $\Theta(1)$
- Insertion – $\Theta(1)$
- Removal – $\Theta(1)$

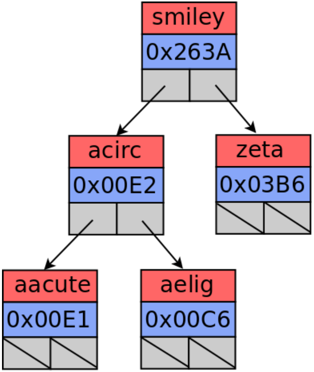
[1] Balanced; random data

Binary Search Tree

- A binary tree is either:
  - The empty tree, or
  - contains a key/value pair, and a left and right subtree, themselves trees
- A binary search tree (BST) has the *sibling order* property
  - The key of a node is greater than all keys in the left subtree
  - The key of a node is less than all keys in the right subtree
- Note, every subtree of a BST is a BST
- O(log n) expected search and insertion time
  - If the tree is balanced
- In-order traversal yeilds keys in sorted order

BST Example

- In the following examples each node stores a key/value pair:
  - key – String, name of the character
  - value – Hexadecimal integer, Unicode encoding
- A reference (pointer) to each of the 2 subtrees



BST in Python

- Let an empty tree be the empty list
- Use a list of size 3:
  1. The key/value pair (another list)
  2. The left subtree
  3. The right subtree
- The following is a tree w/one node:

```
T = [ ['smiley', 0x263A], [], [] ]
```

BST Lookup – Python


```
def lookup( T, name ) :
   '''lookup: look up name in tree T, return the
   cell, None if not found'''

   if T == [] : # T is the empty tree
      return None
   if T[0][0] == name :
      return T
   elif name < T[0][0] : # look in left subtree
      return lookup( T[1], name ) ;
   else : # look in right subtree
      return lookup( T[2], name ) ;
```

BST in C

We will use a struct to hold the key, value and pointers to the subtrees.

```c
typedef struct bNode bNode ;
struct bNode {
   char *name ;
   int value ;
   bNode *left ;
   bNode *right ;
} ;
```

BST in C

```
/* lookup: look up name in tree treep *
 *    Return pointer to node, NULL if not found */

bNode* lookup( bNode *treep, char *name )
{
   int cmp ;

   if( treep == NULL )
      return NULL; /* Didn't find it */
   cmp = strcmp( name, treep->name ) ;
   if( cmp == 0 )
      return treep ;
   else if( cmp < 0 )
      return lookup( treep->left, name ) ;
   else
      return lookup( treep->right, name ) ;
}
```

Hash Table (Open)

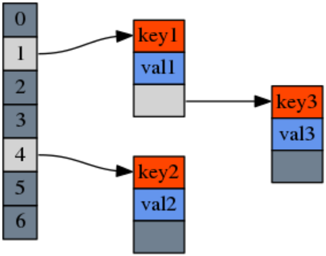
- Provides key lookup and insertion with constant expected cost
- At the heart is a vector with m slots, where it is not usually possible to reserve a slot for each possible element
- Hash function maps key to index (should evenly distribute keys)
  - $H(k, m) \to [0, m-1]$
  - Two keys might have the same has value – *collision*
- Duplicates stored in a chain (list) – other strategies exist

Hash Table (Open)


```c
typedef struct sNode sNode ;

   /* An entry */
struct sNode {
   char* name ;
   int value ;
   sNode* next ; /* in chain */
} ;

   /* The table (array) */
sNode* symtab[NHASH] ;
```



In this example, $key1$ and $key3$ have the same hash value, 1

A Simple Hash Function in C


```c
int MULTIPLIER = 31 ;

   /* hash: compute hash value of string */
unsigned int hash( char* str )
{
   unsigned int h ;
   unsigned char *p ;
   h = 0 ;

   for( p=(unsigned char*) str; *p!='\0'; ++p )
   {
      h = h*MULTIPLIER + *p ;
      h %= NHASH ;
   }

   return h ;
}
```

Hash Table Lookup/Insert in C

```
   /* lookup: find name in symtab, with optional create */
sNode* lookup( char* name, int create, int value )
{
   sNode* sym ;
   int h = hash(name) ;

   for( sym=symtab[h]; sym != NULL; sym=sym->next)
      if( strcmp( name, sym->name ) == 0 )
         return sym ;
   if( create ) {
      sym = (sNode*) malloc( sizeof( sNode )) ;
      sym->name = name ; /* assumed allocated elsewhere */
      sym->value = value ;
      sym->next = symtab[h] ; /* insert at front */
      symtab[h] = sym ;
   }
   return sym ;
}
```