Data Structures

Dr. Axel Kohlmeyer

Assistant Dean for High-Performance Computing
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu

Data Types in Programming

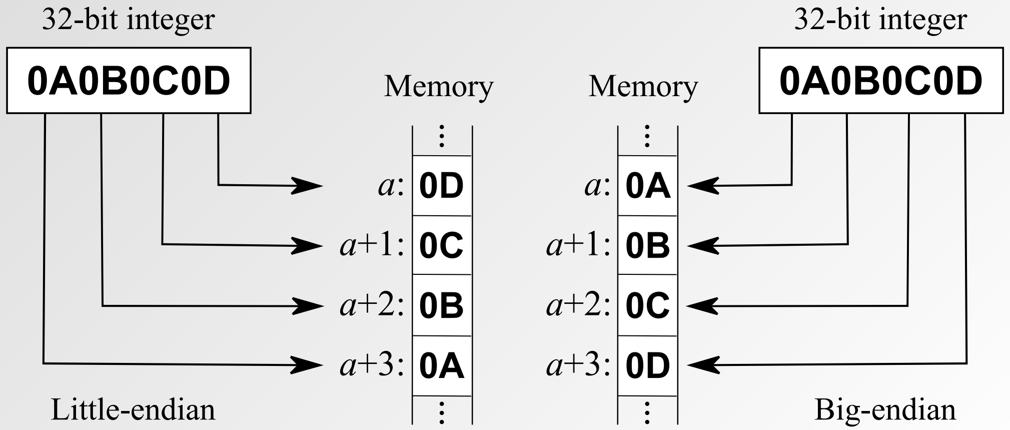
- Three categories: <u>primitive</u> data types, <u>compound</u> data types, and <u>abstract</u> data types
- Primitive data types are the “elementary” data types representing single items. Examples: character, int, float, double, enum, bool etc.
- Compound data types combine multiple primitive data types to a data type. Examples: array, struct (derived type), union, bitfield, string
- Abstract data types implement a more complex behavior. Examples: stack, map, queue, tree

Pointers and References

- A pointer variable is a data type that stores the memory address of a data element; pointers are associated with a data type or generic/void
- To access the associated data, the pointer needs to be dereferenced, different elements can be accessed with “pointer arithmetic”
- Change associated data type of a pointer with a “type cast” → data will be reinterpreted
- A reference is a variable representing a dereferenced pointer → no pointer arithmetic

Big Endian versus Little Endian

- The storage elements of computers are “bits” for registers and “bytes” for RAM
- To represent a primitive data type (a “word” → “register”), multiple bytes may be are needed
- The order of those bits and bytes is specific to the computer architecture → endianness
- Examples:
  - x86 is little endian
  - SPARC, Power AIX are big endian



Master in High Performance Computing

struct, bitfield, union

- struct, bitfield, union are compound data types
- Constructed from multiple primitive data types
- In a struct the elements follow in memory in order of definition, in a union they overlap and share storage (specific layout is undefined)
- A bitfield is a special case of a struct where the number of bits per element is limited

```
struct {                              struct {
    unsigned int negative:1;              unsigned int mantissa:23;
    unsigned int exponent:8;              unsigned int exponent:8;
    unsigned int mantissa:23;             unsigned int negative:1;
} ieee_float_big_endian;              } ieee_float_little_endian;
```

Arrays

- Arrays are compound data structures from identical primitive data types that can be referenced by its index
- Memory storage is consecutive in memory so elements with index +/-1 are next to each other
- In C/C++ indexes start with a 0, in Fortran a 1
- In C/C++ multidimensional arrays are built from 1d arrays: a 2d array is a 1d array of 1d arrays; Storage may be disjointed between 1d arrays
- In Fortran array storage is always 1d ("flat")

Compound Data Structures in C/C++ and Fortran

- The C/C++ language have multiple variants of compound data structures due to the more low level nature of the C language kernel
- Fortran “only” has derived types, which may contain “type bound procedures” (like classes)
- In C++ “struct” and “union” can be used like classes (they have constructors and can contain functions with overloading etc.)
  The class/struct/union prefix is optional.
- The equivalent in C requires a “typedef”

Abstract Data Types Behavior

- Abstract data type implement a more complex behavior and thus use more complex ways to store and access its stored data:
  - Random access → look up data by its index
  - Sequential access → look up the next/previous item
  - Search → look up item by its value
- Operations on data (not all may be available):
  - Insert/delete items at the beginning or end
  - Insert/delete items in the middle
  - Assign new value to given element

Implementing Abstract Data Types

- To implement abstract data types we have to use primitive or compound data types or other abstract data types for storing the data and information about it or how to access it
- Often it is desirable to mimic the interface of simpler data structures like arrays.
  Example: STL containers in C++
- We need specific functions that “manage” the data internally and allow to “get” and “set” data:
  => object oriented programming

Sequential Data Structures

- Sequential data structures store its data in a way that it is possible to iterate it, i.e. loop over all elements in a specific order: the order in which the items have been added.
- Simple example: Array
  - Access to data via index (→ access is O(1))
  - Next item is index+1 (or use pointer arithmetic)
  - Problem: inserting/appending an element requires creating a new array and copying elements over → index for elements changes after inserted item

Linked List

- The linked list avoids the copying or moving or data when inserting an element
- Each storage element is a struct that can store the value and a pointer variable to the next item
- The pointer for the last item is a NULL pointer
- One pointer is needed for the first element
- To insert a new item, allocate new element, assign value, the “next” pointer of the new item is set to the “next” value where it is inserted, and that “next” value will be set to the new item

Linked List – Insert at Head

```
typedef struct _item item_t;
struct _item { float value; item_t *next; };

item_t *head = NULL;
int main(int argc, char **argv)
{   int i;
    for (i=1; i < argc; ++i) {
        item_t *newitem = (item_t *)malloc(sizeof(item_t));
        newitem->value = atof(argv[i]);
        newitem->next = NULL;
        if (!head) {
           head = newitem;
        } else {
           newitem->next = head;
           head = newitem;
        }
    }
    return 0;
}
```

Linked List – Loop over List

```c
typedef struct _item item_t;
struct _item { float value; item_t *next; };

int main(int argc, char **argv)
{   int i;

    /* code to insert items in list here */

    printf("current list is:");
    if (head) {
       item_t *current = head;
       do {
           printf(" %g",current->value);
           current = current->next;
       } while (current != NULL);
    }
    printf(" end\n");
    return 0;
}
```

Linked List – Delete Element

- Find element to delete and get pointer to it while keeping a pointer to the previous element
- If element is “head” of list (“previous” == NULL):
  - Make “next” of the looked up element new “head”
  - Delete the looked up element
- Otherwise:
  - Assign the “next” of the “previous” item to the “next” pointer of the looked up item
  - Delete the looked up element
- Doubly linked list includes “previous” in element

Advantages/Disadvantages of Linked List

- Advantages
  - Insert at either end is O(1)
  - Insert can be at any point with O(1), but lookup and search is O(n)
  - Only allocate memory as needed
  - Can store data in sorted fashion
- Disadavantages
  - Search for a specific element is O(n)
  - Memory is fragmented, extra memory for pointers
  - Can only look up or search in one direction
    → doubly linked list to avoid that issue

Stack, Queue

- A stack is a sequential data structure optimized for adding and removing data at the end (top)
  `stack->push(item);` *or* `item = stack->pop();`
- A stack can be conveniently implemented on top of either an array or a linked list:
  - array: allocate array with extra space, keep index to top of stack, grow array in chunks (by factor)
  - linked list: “head” is top of stack, always insert and delete at “head”
- Queue optimized for push at end, pop at front:
  can use linked list with pointers to head <u>and</u> tail

Associative Data Structures

- Associative data structures do not store elements in a specific sorted sequence
- The index may not be an integral value (e.g. a string)
- Examples: set, map (aka associative array or dictionary), sparse array
- No iterator to access elements in order (instead get list of keys, sort keys, look up items by key)
- Typical to implement this via hash table

Hashing Function

- Hashing function: take arbitrary input (e.g. string) and assign a key out of a fixed size pool of keys. Example: value of first letter of string → Problems: 128 possible values, not evenly distributed (some values more common)
- Optimal hashing function: there is exactly one unique return value for each unique input. That is rarely possible.
- Good hashing function: for any selection of inputs any the return values are equally likely

Simple Hash Algorithms

- Modulus of next prime number: gives a better distribution of hashed values in case multiples of some factors are more frequent among keys => value range: 0 → next prime - 1
- Middle bits of squared key: multiply key with itself and extract n bits from middle of that value => value range: 0 → next power of 2 – 1
- Top bits after multiply with prime: multiply key large prime number and extract n bits from top => value range: 0 → next power of 2 - 1

Hash Table

- A hash table is a data structure for an associative array
  - The underlying storage is an array of “buckets”, which can be linked lists or arrays or stacks etc.
  - The size of this array of buckets is an input
  - The hashing function takes the key and computes an index for a bucket where the data is then added (if the key does not exist) or reassigned (if the key is already present)
  - To look up an element by its key, compute the hash, which is O(1), and then search through the buckets for the specific key, which is O(n)
  - So ideally the number of buckets is chosen so that the number of items in each of them is small, but also that storage is not wasted by having many empty buckets

Advantages / Disadvantages of Hash Table

- Advantages
  - Look up is fast (O(1) unless a bucket gets large)
  - Storage is dense, even for sparse data → no waste
  - Key/index need not be an integer → dictionary
- Disadvantages
  - Data is not stored in order,
    can only iterate in random order
  - Efficiency depends on good hashing function
    → fast and evenly distributed across desired number of buckets

Binary Search Tree

- A binary search tree is an abstract data structure that tries to retain the advantages of linked lists while reducing the disadvantages
- Data is stored so it can be iterated over in order
- Data can be inserted at O(log n) cost, not O(n)
- Data can be looked up at O(log n) cost
- Tree consists of “nodes”, each contain a “key”, a “value” and a pointer to a node on the “left” and “right” on the next level

Binary Search Tree – Add Item

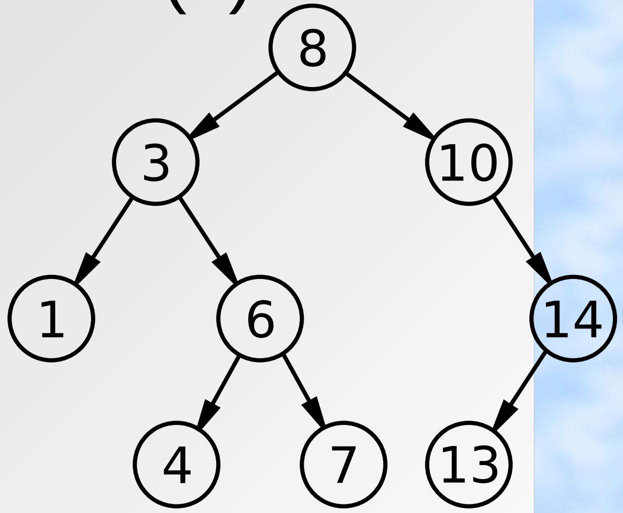
- Allocate node, set key/value, left/right to NULL
- Start with root node: if NULL assign new node
- Otherwise compare key to key of “root” node:
  - If key is smaller, to to left node
  - If key is larger, go to right node
  - If key is the same, reassign value and discard node
  - Repeat until node is NULL then assign with node
- In perfect distribution (= a well balanced tree) the “depth” is log(n), i.e. the number of nodes doubles on each level
- A maximally unbalanced tree is like a linked list

Binary Search Tree – Iterate in order

- Start with root node, move always to left until the left pointer is NULL.
  This is the element with the smallest key
- Now got to right element, if present, and follow left elements again until NULL, this is the element with the next smallest key, otherwise go up (next smallest) and right, if possible
- This allows to traverse the tree in order from the element with the smallest key to the one with the largest key

Binary Search Tree Advantages and Disadvantages

- Advantages
  - Data structured so it can be iterated in sorted order
  - Lookups are (ideally) O(log n) or worst O(n)
  - Nodes are only added, not inserted
- Disadvantages
  - Needs more space than linked list
  - More complex to traverse in order
  - Good performance depends on a balanced tree (better and more complex tree algorithms exist, but can also regularly rebalance the tree)



Binary Search Tree – Delete Node

- Find node to delete and get pointer to it while keeping a pointer to its parent node
- If looked up node is “root” node:
  - Make left leaf new “root” node, if NULL use right
  - If not made “root” insert right leaf as new item
  - Delete looked up node
- Otherwise:
  - Assign left leaf where current node was stored in parent node
  - If not NULL insert right node as new element to tree
  - Delete looked up node
- Simpler if node data structure includes pointer to parent

Simple Strategy to Rebalance a BST

- Start a new tree with a pointer to a root node
- Create ordered array of keys of items in tree
- Pick the key in the center for root of new tree
- Now add process keys to left, then to right
- When working on sub-arrays start with center
- Recurse until sub-arrays have < 2 elements
- Add remaining elements and return
- When new tree is complete delete old one

Data Structures

Dr. Axel Kohlmeyer

Assistant Dean for High-Performance Computing
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu