Notes 2: Introduction to data structures

2.1 Recursion

2.1.1 Recursive functions

Recursion is a central concept in computation in which the solution of a problem depends on the solution of smaller copies of the same problem. Recursion is a conceptually different approach to thinking about numerical algorithms. It stands in contrast to *iteration* which is the more familiar way of thinkin about algorithms (at least to most people with backgrounds in mathematics, physics or engineering). All recursive procedures have an alternative formultion as iterative procedures. We will take only a superficial look at recursion in this course since it provides a very neat way to represent certain useful numerical functions and data structures. Computer Science texts go into much greater depth.

Most modern programming languages support recursion by allowing a function to call itself. A recursive function usually consists of one or more recursive cases (values of the inputs for which the function calls itself) and one or more so-called base cases (values of the inputs for which the function returns a results trivially without recurring). An important consideration when designing recursive functions is to make sure that a base case is always reached for allowed values of the input data. Failure to do this can easily result in non-terminating infinite recursions which can be rather difficult to debug. Consider the following classical example which implements the factorial function as a recursive function:

```
int f(int n)
{
  if (n == 0)                // Base case
  { return 1;
  }
  else
  { return n * f(n-1);       // Recursive case
  }
}
```

If you have never encountered the concept of a recursive function before, it takes some thought to see that $f(n) = n!$. One way to understand what is happening is to write out the set of function executions which occur when the function is called. Consider, for example when we calculate $f(4)$. Using brackets to keep track of the levels of the recursion we have

```
f(4) = 4 * f(3)              // Input 3 calls recursive case
     = 4*(3*(f(2))           // Input 2 calls recursive case
     = 4*(3*(2*f(1))         // Input 1 calls recursive case
     = 4*(3*(2*(1*f(0))))    // Input 0 calls base case returning 1
     = 4*(3*(2*(1*1)))       // Multiplications now go from inside the out
     = 4*(3*(2*1))
     = 4*(3*2)
     = 4*6
     = 24
```

We have already seen some less trivial examples of recursive functions in the "divide-and-conquer" algorithms introduced in Secs. 1.2 and 1.3. Recursion is the natural framework for thinking about divide-and-conquer algorithms. If $F(n)$ represents the number of FLOPs required to solve a problem of size n, we have seen that for divide-and-conquer algorithms, $F(n)$ satisfies a recurrence relation. In the literature on the analysis of the computational complexity of algorithms, the so-called "Master Theorem" provides asymptotic bounds on the complexity of recursively defined divide-and-conquer algorithms. Here we just provide the statement for the purposes of application.

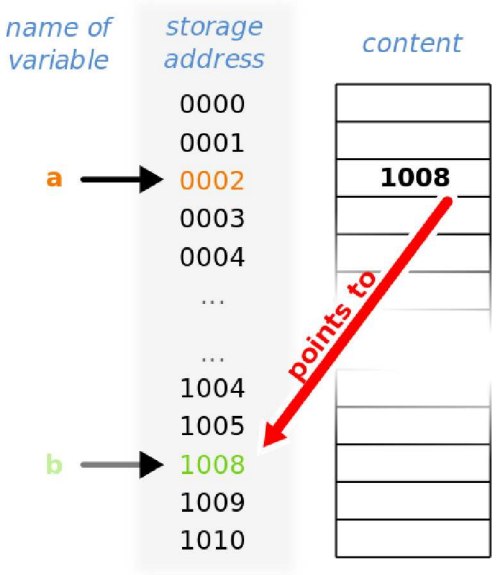


Figure 2.1: (from wikipedia) Graphical illustration of pointers. The value stored in the pointer shown is the address of the variable b.

Theorem 2.1.1. *Let a be an integer greater than or equal to 1 and b be an integer greater than 1. Let c be a positive real number and d a nonnegative real number. Given a recurrence of the form*

$$F(n) = \begin{cases} aF(n/b) + n^c & \text{if } n > 1 \\ d & \text{if } n = 1, \end{cases}$$

then if n is a power of b there are three cases:

1. *if $\log_b(a) < c$ then $F(n) = O(n^c)$*
2. *if $\log_b(a) = c$ then $F(n) = O(n^c \log(n))$*
3. *if $\log_b(a) > c$ then $F(n) = O(n^{\log_b(a)})$.*

You should check whether this result applies to the examples with have seen already. For more information on recurrence relations and a proof of the Master Theorem see [1]

2.2 Linked lists

2.2.1 Structural recursion

A data structure is a particular way of organizing data in a computer so that it can be used efficiently. Simple examples of data structures are linear arrays (which allow a list of data to be stored and accessed without assigning a different variable to store each element of the list) or a struct object in C which allows variables or objects of several different types to be grouped together into a single structure which can be easily manipulated inside a code. Some more sophisticated and powerful data structures emerge when the concept of recursion, which up until now we have thought of as a property of a function, is applied to a data type. The resulting *recursive data types* are data structures which are defined in terms of themselves. This is sometimes referred to as "structural recursion". Examples include structures like linked lists, stacks, queue and trees which facilitate very efficient implementation of certain numerical procedures.

2.2.2 Pointers

Before studying recursive data structures it is helpful to understand the concept of a pointer. Pointers are one of the most powerful features of modern programming languages like C/C++ or Fortran90. They are also one of the easiest features to use incorrectly, often resulting in bugs which are highly nontrivial to find. As a result of the potential for accidental (or deliberate) mis-use, some programming languages (such as Java and MatLab) don't support pointers natively although they provide complex data types which implement pointer-like functionality with additional checking to prevent mis-use.

In simple terms, a pointer is a type of variable which stores the address of ("points to") a regular variable. See Fig. 2.1. A pointer references a location in memory. In order to be useful, it is usually necessary to read the value of the variable which

a pointer points to. The operation of obtaining the value of a variable referenced by a pointer is known as *dereferencing* the pointer. Most of the applications of pointers exploit the fact that it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.

In C/C++ the syntax for defining and dereferencing a pointer involves the symbol *. The syntax for obtaining the address of a variable to store in a pointer involves the symbol &. Here is an example:

```
int a = 6;              // Define an int a and set its value to be 6
int *ptr = NULL;        // Define a pointer ptr and set its value to be NULL

ptr = &a;               // Set ptr to point to a
cout << *ptr << endl; // Print the value of the variable pointed to by ptr
```

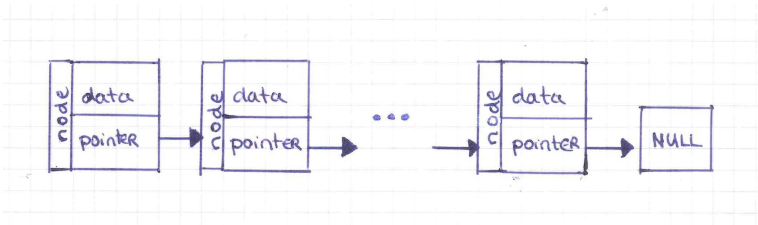
The value NULL has a special meaning in the C/C++ language. A NULL pointer has a value reserved for indicating that the pointer does not refer to a valid object.

The power and dangers of using pointers are associated with the fact that they can be directly manipulated as numbers. As a result they can be made to point to unused addresses or to data which is being used for other purposes. This can lead to all sorts of programming errors. A pointer which does not have any address assigned to it is called a "wild" pointer. Any attempt to use such uninitialized pointers can cause unexpected behavior, either because the initial value is not a valid address, or because using it may damage other parts of the program. The result is often a segmentation fault or other storage violation. Even worse, in systems with explicit memory allocation (in C this is implemented using the malloc() function and its variants), it is possible to create a "dangling" pointer by accidentally deallocating the memory region it points into. This type of pointer is dangerous and subtle because a deallocated memory region may contain the same data as it did before it was deallocated but may be then reallocated and overwritten. This can lead to the most evil types of "random" bugs in which a program sometimes runs correctly (if the memory pointed to by the dangling pointer happens not to be overwritten before it is used) but sometimes runs incorrectly or crashes with a segmentation fault (if the memory pointer to by the dangling pointer is reallocated for some other purpose before it is used).

Let us now look at how pointers can be used to do useful things.

2.2.3 Linked lists

A linked list is a data structure which contains a contains a sequence of nodes. In the simplest type of linked list (the so-called singly linked list), each node consists of some data and a pointer to the next node in the list. Usually the last node in the list has a NULL pointer indicating the end of the list:



In some respects a linked lists is like a conventional linear array in the sense that is stores a sequence of data objects. It has some advantages over a conventional linear array however:

- Linked lists are dynamic data structures. The memory is allocated as the items are added to the list so it is not necessary to know the length of the list in advance.
- Elements can be efficiently inserted or removed from the list without reallocation or reorganization of the rest of the elements. This is because the list elements do not need to be stored contiguously in memory.
- They form the building blocks of more complicated data structures like trees and networks which are not easily implemented as linear arrays.

There are some disadvantages too:

- The biggest disadvantage is that linked lists only provide sequential access to list elements. To get to node i requires traversing the list from the beginning. This makes direct random access to list elements very inefficient.
- Because each node stores a pointer in addition to an item of data, a linked list takes up more memory than a linear array to store an equivalent amount of data.

Since each node of the list points to another node which points to another node etc, it is natural to define a linked list as a recursive data structure: a linked list is either empty, or it is a node which points to a linked list. Here is a simple C++ class which implements such a definition for a linked list which stores integer data:

```
class List
{
   int n;                          // The data is just an int
   List *next;                     // Pointer to the next node in the list
};
```

We can add to this class a method which, given a pointer to a list and an integer, inserts a new item at the head of the list and then returns a pointer to the modified list:

```
List * List::insertHead(List *list, int N)
{
   List *new_node = new List();     // Create a new node to be the new head
   new_node->n = N;                 // Assign the data to the new node
   if(list != NULL)
   {  new_node->next = list;        // New node points to the previous list
   }
   return new_node;                 // Return a pointer to the new list
}
```

If you can understand this piece of code then you *really* understand pointers. Let us now add a method which, given a pointer to a list and an integer, inserts a new item at the *tail* of the list and then returns a pointer to the modified list. One way to do this is via recursion:

```
List * List::insertTail(List *list, int N)
{
   if(list == NULL)                 // Base case
   {
      List *new_node = new List();  // Create a new node
      new_node->n =N;               // Store the data
      return new_node;              // Return the new node
   }
   else                             // Recursive case
   {                                // Recurse on list->next
     list->next = insertTail(list->next, N);
     return list;
   }
}
```

If you can understand this piece of code then you *really* understand recursive functions. Does this code really insert a new node at the tail of the list? Consider the following argument:

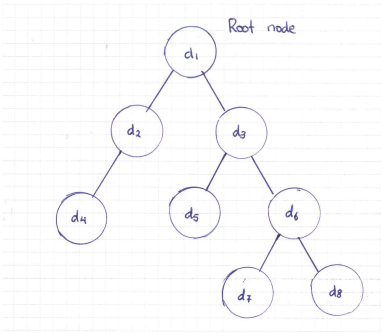
- If list has length 0, the base case holds. The method returns to a list consisting of only a new node containing the integer N. This is the correct answer.
- If list has length $n > 0$ then the recursive call is applied to the List list.next which has length $n - 1$.
- Let us assume that this recursive call gives the correct result for a list of length $n - 1$. That is to say we assume the recursive call returns a pointer to the List list.next (containing all nodes after the first) with a new node containing the integer N inserted at its rear. Given this assumption, storing a pointer to this List into list->next and returning list produces the correct result for the list of length n.
- By induction, we therefore obtain the correct result for a List of any finite length.

Many operations on linked lists are most easily written using recursive methods. This is natural, because linked lists are themselves defined recursively. A basic implementation of the above class can be downloaded from the class website for you to play around with.

2.3 Binary trees

It is fairly easy to implement linked lists in alternative ways to the approach described in Sec. 2.2 which avoid recursion. Some more complicated data structures however, are very difficuilt to implement without recursion. One of the simplest and most useful of such structures is a binary tree. A binary tree is a tree data structure in which each node has at most two

children, which are referred to as the left child and the right child. The top node in the tree is called the root node. As with linked lists, a binary tree data structure usually contains some data at each node in addition to pointers to the left and right child nodes. Here is a sketch of a small tree with 8 nodes storing the list of data items $\{d_1, d_2, \ldots d_8\}$ in "breadth-first" search order :



Such a structure is defined recursively as follows:

```
class BinaryTree
{
  Datum data;
  BinaryTree *L;
  BinaryTree *R;
};
```

Here the Datum object represents a custom defined container object which can contain whatever we want.

2.3.1 Binary tree search

Binary trees have useful applications in search and sorting problems. One particularly important application of trees in search is a structure called a *binary search tree*. Consider an array of N key-value pairs, $\{(k_1, d_1), (k_2, d_2), \ldots (k_N, d_N)\}$ which can be sorted on the key values. This is to say, the list can be arranged so that $k_1 \leq k_2 \leq \ldots \leq k_N$. For a random key value, k, between k_1 and k_N we wish to return the data value associated with the key k. The naive approach would be to simply iterate through the sorted list one by one, comparing the key of each list element with k until we either find one that matches or reach a value which exceeds k (in which case the key k is not in the list). On average this is clearly an $O(N)$ operation. By storing the list in a binary search tree, this operation can be accomplished in $O(\log(N))$ time. A binary search tree, sometimes also called an ordered or sorted binary tree, is a binary tree with a key-value pair at each node withoutthe special property that the key of any node is larger than the keys of all nodes in that node's left child and smaller than the keys of all nodes in that node's right child. A small modification of the above incorporates the search key (we take the key to be an integer) :

```
class BinaryTree
{
  int key;
  Datum data;
  BinaryTree *L;
  BinaryTree *R;
};
```

Fast search is achieved by descending through the tree from root to leaves. At each level, a single comparison with the node key determines whether to go left or right. The total number of comparisons required to reach the leaf level is obviously equal to the number of levels which (assuming that the tree is "balanced") scales as $\log(N)$. We begin by examining the root node. If the tree is null, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the value at that node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is null. If the searched key is not found before a null subtree is reached, then the item must not be present in the tree. This is easily expressed as a recursive algorithm which returns a pointer to the node containing the matching key (or the NULL pointer in the case that the key is not matched):

```
BinaryTree * search(BinaryTree *T, int k)
{
  if (T == NULL)
  {   return NULL;  // base case
  }
  else if (T->key == k)
  {   return T;
  }
  else if (T->key < k)
  {    return search(T->L, k);
  }
  else if (T->key > k)
  {    return (T->R, k);
  }
}
```

Let us use Theorem 2.1.1 to verify our expectation that binary tree search should be an $(O(\log(N))$ operation. Assuming that the tree is balanced, both the left and right child trees will have approximately half the number of nodes. The amount of work which we have to do at each iteration is therefore (at most) 4 comparisons followed by a search on a tree of half the size. Mathematically

$$F(n) = F\left(\frac{n}{2}\right) + 4,$$

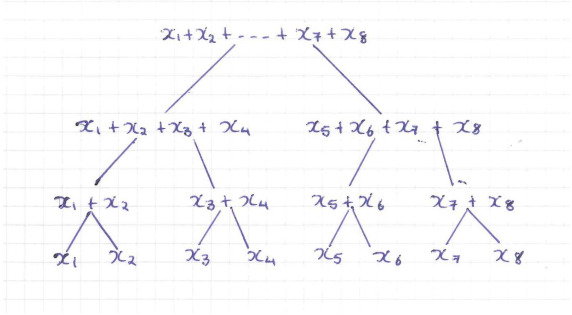
giving $a = 1$, $b = 2$ and $c = 0$ in the notation of Theorem 2.1.1. We see that $\log_b(a) = \log_2(1) = 0 = c$. We are therefore in case 2 and $F(n) = O(\log(n))$.

2.3.2 Searching lists of partial sums - Fenwick trees

The following problem arises so frequently in simulation of stochastic processes (see [2]) that it is worth devoting a little time to study how to solve it efficiently. We are given a list of pairs, $\{(x_1, k_1), (x_2, k_2), \ldots (x_N, k_N)\}$, in which each x_i is a real number and each k_i is an integer. In the application to stochastic simulation, we have a list of possible events which can occur in the system labelled $k_1, \ldots, k_N$ and an associated list of rates $x_1, \ldots, x_N$, at which these events occur. From the list of rates we can calculate the list of partial sums,

$$\{s_i : i = 0, 1 \ldots N\} \qquad \text{where } s_i = \sum_{j=1}^{i} x_j,$$

(we take $s_0 = 0$). The computational challenge is the following: given a number $x \in (0, s_N]$, can we efficiently find i such that $s_{i1} < x \leq s_i$. Essenatially we want to know in which interval of the list of partial sums does the value x lie. In applications to stochastic simulation we usually want to return the index k_i since this determines which event happend next. Obviously we can solve this problem by linear search on the sorted list of partial sums. One average this would require $O(N)$ comparisions. In fact we can use the ideas of Sec.2.3.1 to design a data structure which can solve this problem in $O(\log(N))$ time. The required data structure is a binary tree in which each node stores the sum of the x values of all the nodes in its left and right children. Here is what it looks like for the sequence of values $x_1, x_2, \ldots x_8$:



Such a structure is called a Fenwick tree. If we leave aside the question of how to build the tree starting from the initial list, the process of searching for the interval containing a particular value x is rather similar to the binary tree search. We

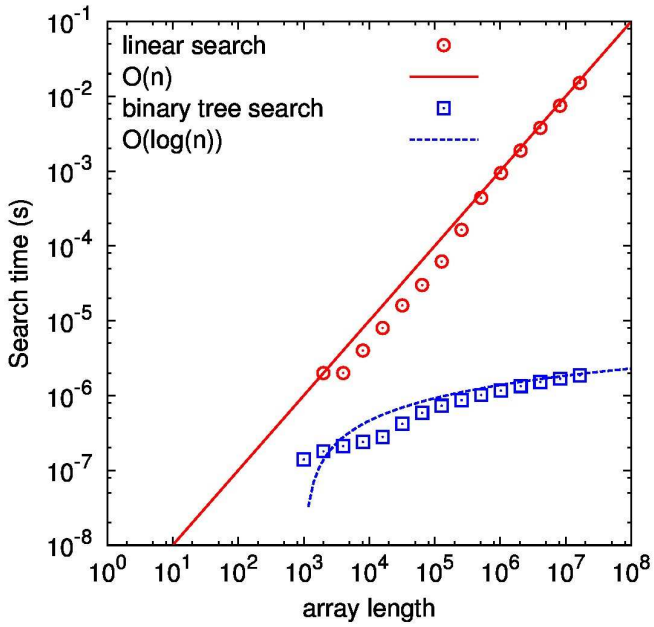


Figure 2.2: Comparison of the performance of binary tree search (blue squares) and linear search (red circles) for the problem of searching the running total at an array of real numbers.

compare x to the sum of the values stored in left child and if x is less than this value then we search for x on the left branch. If not, then we subtract from x the value of all the nodes on the left branch and search for the result on the right branch. The subtraction is the bit which is easy to get wrong here. Check the diagram above to convince yourself that this is correct. When we get to a leaf node then we return the data stored there. Here is a recursive implementation of this process in C++

```
Datum BinaryTree::search(BinaryTree *T, double x)
{
  if(T->L==NULL && T->R==NULL)
  { // Base case: we have reached a leaf node. Return the data stored there
    return T->data;
  }
  else
  {
    // Recursive case: decide whether to go left or right.
    double y=(T->L->data).x;  // Get the sum of the left child
    if(x <= y)
    { // Search left child
      return search(T->L, x);
    }
    else
    { // Search right child
      return search(T->R, x-y);
    }
  }
}
```

When doing large scale stochastic simulations, the extra work associated with implementing such a data structure is well worth the increase in speed of the resulting code. Fig. 2.2 shows the results of doing some performance measurements of search on a Fenwick tree compared to naive linear search. If you have not used binary search algorithms in your programming before, the increase in speed for large arrays is truly awesome to behold! An implementation of a Fenwick tree is available from the class website for you to play around with. In particular, this implementation contains a recursive

function which builds the tree in the first place. Building the tree is an $O(N)$ operation. Another important point about Fenwick trees and binary search trees in general which makes them practical for efficient implementation of Monte Carlo simulations is that once a tree has been built, the operations of insertion, deletion and updating of nodes can also be done in $O(\log(N))$ time. This means that it is not necessary to rebuild the tree each time the underlying array of rates changes. A long sequence of insertions and deletions can however result in the tree becoming unbalanced resulting in a loss of the fast search property as the number of levels grows (the extreme case of a completely unbalanced tree has N levels and search becomes equivalent to linear search). In practice, the tree may need to be rebalanced at occasional intervals. These are more specialise topics which we will not cover in this course but it is good to be aware of them if you end up using these methods in your research.

Bibliography

[1] K. Bogart and C. Stein. CS 21/math 19 - course notes. `https://math.dartmouth.edu/archive/m19w03/public_html/book.html`, 2003.

[2] J. L. Blue, I. Beichl, and F. Sullivan. Faster monte carlo simulations. *Phys. Rev. E*, 51(2):R867–R868, 1995.