ECE 264 Advanced C Programming

Contents

1 Complexity

In a binary search tree, how many nodes do we need to check in order to determine whether a value is currently stored in the tree (the `BTree_search` function)? Let's ask a different question first. How many nodes does a tree have if the tree has n levels? In the first level (the root), there is only one node. The next level has at most two nodes. The third level can have up to four nodes.

level	node
1	1
2	2
3	4
4	8
...	...
n	2^{n-1}
total	$2^n - 1$

If a binary search tree is *complete* and *balanced*, searching a tree of $2^n - 1$ nodes requires visiting at most n levels. We can eliminate half of the remaining nodes in each step. To put it in another way, searching a tree of n nodes requires visiting only $\approx \lg n$ nodes; here $\lg$ is the logarithm function using 2 as the base, namely $\lg n = \log_2 n$. We usually write $\log$ using 10 as the base, $\log n = \log_{10} n$.

If we use a binary search tree to sort a sequence of numbers, what is the complexity? What do we mean by using a tree to sort? We can insert these numbers one by one and then use

in-order to print the numbers. If the tree is always balanced after inserting $i-1$ numbers, inserting the i^{th} number has to visit only $\approx \lg(i-1)$ nodes. Therefore, the complexity is

$$\approx \sum_{i=2}^{n} \lg(i-1). \tag{1}$$

Let's simplify it by finding an *upper bound*. We know $\lg x < \lg y$ if $1 < x < y$. Thus,

$$\sum_{i=2}^{n} \lg(i-1) < \sum_{i=2}^{n} \lg(n) < n \lg n. \tag{2}$$

Sorting n numbers using a binary search tree requires $n \lg n$ steps.

This is much better than n^2 using a linked list. This assumes that the tree is always balanced. In reality, we may not be that lucky. Let's consider the situation when the sequence is already sorted in descending order. When we insert the numbers, only the left subtrees are used and inserting the i^{th} number has to visit $i-1$ nodes. As a result, it takes $\approx n^2$ to sort the numbers. Similarly, if the numbers are sorted in ascending order, only the right subtress are used and the complexity is also n^2. For the **average** cases, we assume the numbers are not ordered and the tree is "somewhat" balanced, the complexity is $n \lg n$. The detailed analysis is beyond the scope of ECE 264.

2 Design Fast Algorithms

Why is it faster to use a binary search tree to sort than to use a linked list? This reveals a basic principle in designing faster algorithms:

avoid redundant computation.

Why is it slower to compute Fibonacci numbers by using

```
int Fibonacci(int n)
{
  if (n == 0) { return 0; }
  if (n == 1) { return 1; }
  return (Fibonacci(n - 1) + Fibonacci(n - 2));
}
```

We compute $f(k), 1 \leq k < n$ many times. We should remember what we have done and do not repeat the computation.

When we sort numbers using a linked list, we repeat some computation again and again. The list is already sorted. Why do we start from the very beginning for every insertion? Why can't we *jump* to somewhere that is closer to the final location of the new number? That is the idea of a binary search tree. If the new number is larger than some numbers already in the tree, it is unnecessary to compare with these numbers one by one.

3 Quicksort

If we can sort numbers using a binary search tree in $n \lg n$ steps, can we sort even faster using an array? Remember that in an array, we can retrieve any element in a single step. In a binary tree (or a linked list), we have to visit nodes one by one. There are many ways to sort numbers sorted in an array using $n \lg n$ steps. Here, we introduce a method called *quicksort*.

```
#include <stdio.h>

void swap(int *a, int *b)
{
  int t = *a;
  *a = *b;
  *b = t;
}
void sort(int arr[], int beg, int end)
{
  if (end > beg + 1)
    {
      int piv = arr[beg];
      int l = beg + 1;
      int r = end;
      while (l < r)
        {
          if (arr[l] <=  piv)
            {  l++; }
          else
            { swap(&arr[l], &arr[--r]); }
        }
      swap(&arr[--l], &arr[beg]);
      sort(arr, beg, l);
```

```
      sort(arr, r, end);
    }
}

int main(int argc, char * argv[])
{
  int data[] = {5, 4, 3, 9, -1, 0, 4, 3, 2, 11, 7, 6, 8, 9, 14};
  int length = sizeof(data)/ sizeof(int);
  int index;
  sort(data, 0, length);
  for (index = 0; index < length; index ++)
    {
      printf("%d ", data[index]);
    }
  printf("\n");
  return 0;
}
```

The basic concept is the same: avoid redundant work. C has a built-in function for quick-sort. In the lecture for March 11, we already used it:

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
int search1(int data[], int size, int value);
int search2(int data[], int size, int value);

void print(int data[], int size)
{
  int ind;
  for (ind = 0; ind < size; ind ++)
    {
      printf("%d ", data[ind]);
    }
  printf("\n");
}

int compare(const void * p1, const void * p2)
/* comparison function needed by qsort */
{
  int v1 = * (const int *) p1;
  int v2 = * (const int *) p2;
  if (v1 > v2) { return 1; }
```

```
  if (v1 < v2) { return -1; }
  return 0;
}

int main(int argc, char * argv[])
{
  int numElement = 0;
  srand(time(0));
  if (argc > 1)
    {
      numElement = (int)strtol(argv[1], (char **)NULL, 10);
    }
  if (numElement < 100) { numElement = 100; }
  int * data = malloc(numElement * sizeof(int));
  int ind;
  /* initialize the elements */
  for (ind = 0; ind < numElement; ind ++)
    {
      data[ind] = numElement - ind;
    }
  print(data, numElement);
  int value = rand() % (5 * numElement);
  printf("search %d, index = %d\n\n", value,
         search1(data, numElement, value));
  qsort(data, numElement, sizeof(int), compare);
  /* qsort (quicksort) is provided by C but we have to provide a
     comparison function. Please check the manual for details */
  print(data, numElement);
  printf("search %d, index = %d\n", value,
         search2(data, numElement, value));
  /* The index may be different from search1 because the elements have
     been sorted. */
  free (data);
  return 0;
}
```

To use C's `quicksort` function, we have to provide a compare function. This allows us to use `quicksort` for sorting strcutres.