MPCS 51040 – C Programming

Lecture 9 – Sorting & Searching, Parallel Programming

Dries Kimpe

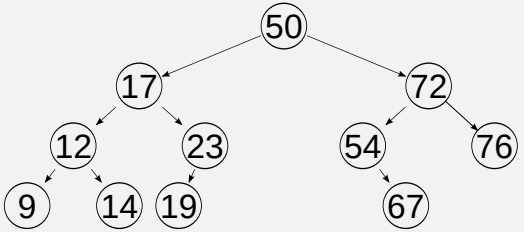
May 22, 2017

AVL Tree

Adelson-Velsky and Landis

Definition

An AVL tree is a *self balancing* binary search tree, which has an additional restriction(property) that, for any node in the tree, the difference between the height of the subtrees *of that node* is at most 1.

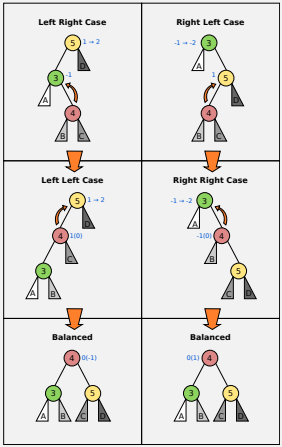


Because of the strict balancing:

Lookup	$\mathcal{O}(log(n)$ worst (and avg) case
Insert	$\mathcal{O}(log(n)$ worst (and avg) case
Delete	$\mathcal{O}(log(n)$ worst (and avg) case

Typically, the balance factor (i.e. difference between the subtree heights) is stored in each node.

AVL Tree Rotations

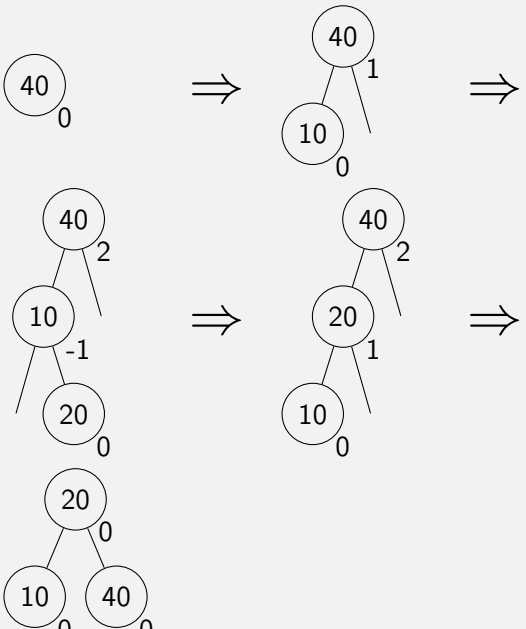


Any operation modifying the tree will be followed by one or more *rotations* in order to restore the AVL tree requirements.

- 4 kinds Left-Left, Right-Right, Left-Right, Right-Left
- To easily determine if the AVL property is violated, each node keeps track of the difference between the heights of its children.
  (For example: left child height - right child height)
- To determine which rotation(s) need to be performed, we look at the *relative position of the just inserted node to the first parent where the balance factor is* ± 2

AVL Tree Rotations

Example



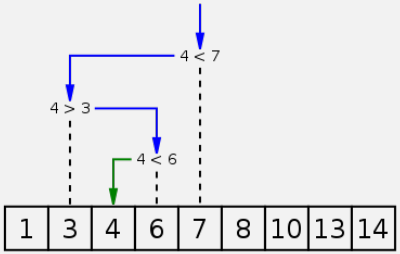
- Adding two elements
- No balance problem yet; for all nodes: $-1 <= \text{the balance factor} <= 1$
- Adding one more element. Violation for the root
  - left child height = 2
  - right child height = 0
- Left-Right case since the new node (20) is the left-right child of the now unbalanced node.
- Left-Right case: need to transform into Left-Left case
- Now transformed into Left-Left case.
- AVL property still violated:
  - left child height = 2
  - right child height = 0

Searching
Definitions

Definition

searching	Locating an element in a data set
linear search	Search all elements of the set until found; $\mathcal{O}(n)$
binary search	(on sorted sets) use the fact that the data is sorted to search in $\mathcal{O}(\log n)$.

Searching
Binary Search



- Same principle as search in BST
- However, in this case, **worst case** $\mathcal{O}(\log n)$
- Repeatedly compare the middle element against the element being searched for.
  - If smaller: repeat for the left half of the data
  - If larger: repeat for the right half.
  - If empty set: not found
  - if equal: element found
- Requires already sorted random-accessible collection (such as sorted array)

Write a *recursive* function implementing binary search in a given array

Definition

Sorting arranging a set of elements in prescribed order.

Comparison-sort Compares elements against each other. Fastest possible is $\mathcal{O}(n \log n)$

Linear-time sort Does not compare elements but instead relies on assuming certain properties of the data: $\mathcal{O}(n)$; Example:

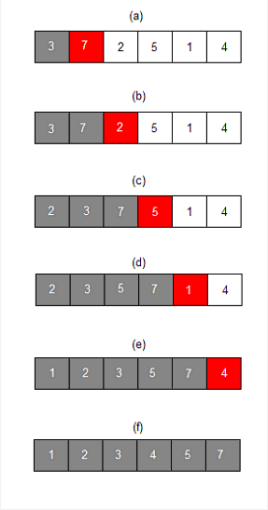
- Sort elements of set $0, 4, 2, 1, 3$ when it can be assumed that all elements are present and there are no duplicates

in-place sort The input data to the sort function is modified and no extra (other than $\mathcal{O}(1)$) space is required.

stable sort Maintains the ordering of elements that are already sorted.

Insertion Sort

Insertion Sort is an *in-place*, *stable* sorting algorithm which has $\mathcal{O}(n^2)$ time complexity. When doing incremental inserting in an already sorted set, the time complexity is $\mathcal{O}(n)$.



- In place sort
- Complexity: $\mathcal{O}(n^2)$
- Very useful when inserting in an already sorted list: $\mathcal{O}(n)$

Algorithm:

- Start on the left of the array (current position = 0)
- Consider the element at the current position.
- Starting from the beginning of the array, move to the right up to the current position inserting the element as soon as it is smaller than the element at that position.
  - Copy elements to the right to make space for at that position.
- Increment current position and repeat.

QuickSort

Overview

QuickSort is an *in place*, recursive, (non-stable in default implementation) general sorting algorithm. Given a set of elements (in array or other random access method):

Worst case performance: $\mathcal{O}(n^2)$.
Average case performance: $\mathcal{O}(n \log n)$

Steps

To sort an array from index $i \cdots j$:

1. Pick pivot element at index k, $i \leq k \leq j$
   (i.e. the pivot is an element of the array)
2. Partition; Move all elements smaller or equal to the pivot element to the left of the pivot element. Move all elements larger than the pivot element to the right of the pivot element.
3. Recursively apply procedure to the partitions $i \cdots p-1$ and $p+1 \cdots j$, where p is the new index of the pivot element. (Note that p is the correct position in of the pivot element in the final sorted array!)

Quicksort

How to partition?

- Pick a pivot element (has to be part of the data)
- Move from the right end of the array towards the left end until an element if found which is less than or equal to the pivot element.
- Move from the left end of the array to the right end until an element is found which is larger than or equal to the pivot element.
- Swap them.
- Continue until both fronts meet.

Note that the pivot element is now in the correct sorted position in the array...

The performance of quicksort depends on picking a good pivot value;

- The $\log n$ in $\mathcal{O}(n \log n)$ originates in finding a pivot which results in *balanced* partitions
- Consider what happens if we always pick the smallest or largest element as pivot... (hint: $\mathcal{O}(n^2)$)

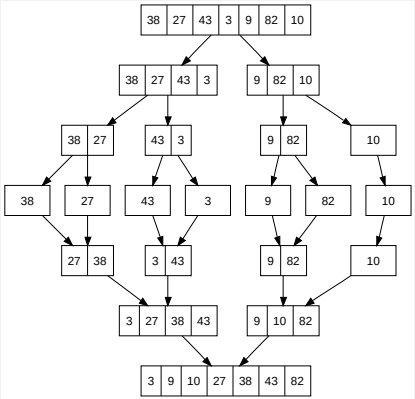
But how to pick a good pivot without looking at all elements?

(Would als result in $\mathcal{O}(n^2)$)

- Option: pick a random element from the set.
- Option: better: pick the middle of element of a selection of 3 randomly chosen elements. (this strongly decreases the changes of picking a 'bad' element)

Quicksort is a popular sorting algorithm because it is very unlikely to show worst case performance.

MergeSort



MergeSort is an 'out of place', usually stable (depends on implementation), recursive sort algorithm with average case $\mathcal{O}(n \log n)$ complexity.

- Split the data in two equal sized portions.
- Recursively sort them using merge sort.
- Merge them (in $\mathcal{O}(n)$) time.

Drawback: Merge sort requires $\mathcal{O}(n)$ extra space.

Counting sort is a linear sort method and assumes a set of n integers. The largest integer in the set is assumed to be $k-1$. The complexity of counting sort is $\mathcal{O}(k+n)$. Counting sort is *stable*.

- Allocate an extra array (counts) of size k holding integers indicating how many times each element occurs.
- Go over the set once, using each integer as an index into the count array. For every set element, increment the corresponding count array element.
- Now for counts, add each element to the next element.
- Scan the input set *backwards* and for each element, look up the corresponding count in the count array and copy the element to that position in a temporary array.
  (why scan backwards?)

- Sorts data in pieces (*digits*) one piece at a time (least significant to most significant)
- Any datatype that can be broken up into 'integer pieces' can be sorted
- Need a *stable* sort for sorting the digits. Why?

Note: not limited to sorting numbers; As long as we can break up the sort key into similar sized items which have numerical represention, we can use radix sort.

Complexity: $\mathcal{O}(pn + pk)$, with p the number of positions (digits) in the input data, n the number of input items, k the radix.

Parallel Programming?

C is an imperative programming language: we provide the compiler with *statements* to tell it what to do. Statements are executed in a well-defined order (compiler can optimize but needs to preserve the illusion of that sequential ordering).

Sequential

Within a single scope (function), statements are executed *sequentially*.

Parallel Computing Many calculations(operations) are performed *simultaneously*.

Parallel Programming Programming a parallel computer.

Almost all current systems are parallel computing systems: most processors are parallel, at various levels (explicitly: multiple hardware threads(cores) or implicitly (superscalar processor, i.e. multiple instructions concurrently).

Parallel Computing

Definition

Parallel computing: performing many calculations (instructions) simultaneously.

Parallelism?

Many forms exist:

- bit-level parallelism
- instruction-level parallelism
- data parallelism
- task parallelism

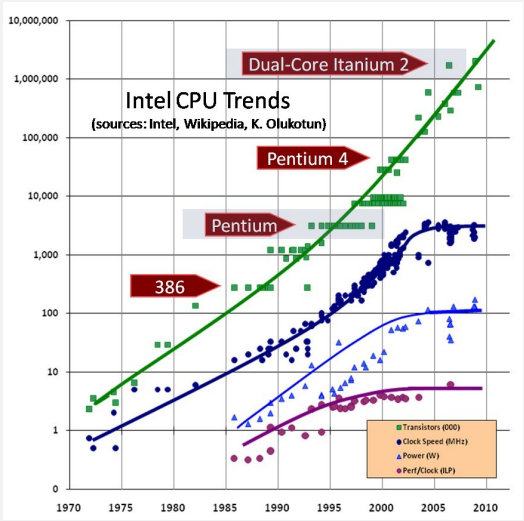
Most forms require explicit programmer effort (except for maybe bit-level and instruction-level parallelism).

Programming for parallel systems requires additional consideration and skills:

- finding and exploiting parallelism
- synchronizing where needed (but not more)

Parallel Programming

If it's harder, why bother?



Why bother with parallel computing?

- Achieving single-threaded gains is more and more difficult
  - Clock frequency difficult to increase (without significant overhead/tradeoff)
  - Complexity already high: branch prediction, out-of-order execution, speculative execution, ...
- Power usage/efficiency is becoming more important (e.g. intel turboboost)

Easier and more efficient to increase parallelism (multiple cores)

NOTE

Usually requires software changes!

Parallel Programming

Some definitions

Definition

Analyzing execution efficiency:

serial No parallelism (one after another)

speedup $S(p) = \frac{T_1}{T_p}$, where T_1 is the time for the *serial* program, and T_p is the time needed to execute the parallel program using p cores.

linear speedup $S(x) \approx x$

superlinear speedup If $S(x) > x$

Processes and Threads

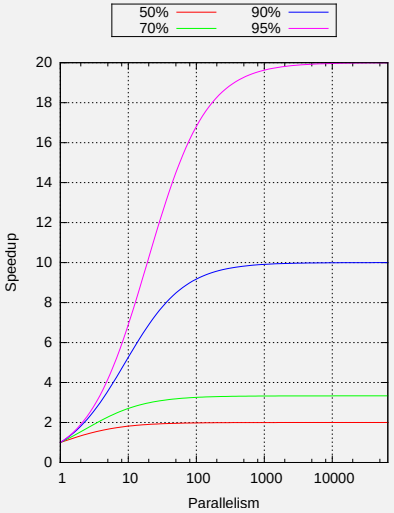
process A program being executed. The process owns the address space (memory), resources (open files etc.), ...

thread A single execution context executing program code. A process has at least one thread.

thread-safe A *thread-safe* function can be called concurrently from multiple threads and continue to function correctly.

Amdahl's Law

Analyzing Performance Potential



Amdahl's law describes, *for a fixed size problem*, the *maximal potential* speedup that can be expected:

$S(x) = \frac{1}{1-p+\frac{p}{x}}$, with p the percentage of the task that benefits from parallelization and x the number of cores used to solve the problem.

In other words, given unlimited parallel processing power, the part which can be parallelized completes in near-zero time, meaning the *serial* time will dominate the overall time.

Notes:

- Assumes serial part remains the same with changing x
- Note logscale (and diminishing returns!)
- Gustafson's law: similar but for increasing problem sizes

$$S(x) = 1 - p + sp$$

Flynn's Taxonomy

SIMD	Single Instruction Multiple Data (stream)
MIMD	Multiple Instruction Multiple Data
SISD	Single Instruction Single Data
MISD	Multiple Instruction Single Data

These are on the *CPU instruction level*. A similar classification can be made at the program (language) level.

SPMD	Single Program Multiple Data
MPMD	Multiple Program Multiple Data

Parallel Programming

Many ways of classifying parallel programming models:

Based on execution model:

- SPMD Single Program Multiple Data
- SPSD Single Program Single Data
- MPSD Multiple Program Single Data
- MPMD Multiple Program Multiple Data

(you might also see 'Instruction' instead of 'Program')

Based on communication model:

- Shared memory
  example: OpenMP
- Distributed memory (message passing)
  example: MPI, go

Parallel Programming & The C language

Support for concurrency in C11:

- C11 adds *optional* support for multi-threaded parallel programming.
- Very few compilers support it.
- Includes defining the **memory model** in the presence of threads.

However, C was used for multi-threaded programming long before C11. For example, POSIX Threads (see `pthread.h`). The required memory model was assumed.

The C11 thread support functions are heavily modeled on posix threads. For now, recommended to use the posix threads functions due to more widespread availability and due to the optional nature of the C11 standard functions.

C11 and Parallel Programming

C11 describes a header (`threads.h`) which provides functions for creating, destroying and synchronizing threads.

Thread support is *optional* in C11, and most compilers do not currently provide C11 thread functionality.

- This is mostly since there have been non-C11 standard specified methods of using threads. For example: POSIX threads (pthreads) described by POSIX.1c standard are available on most common platforms (including Windows, Linux, OS X, Solaris, OpenBSD)

We will be using pthreads (`pthreads.h`)

Review '`man pthreads`', in particular the section 'Thread-safe functions'.

POSIX Threads

```
1  void * PrintHello(void *threadid)
2  {
3     long tid;
4     tid = (long)threadid;
5     printf("Hello World! It's me,"
6            "thread #%ld!\n", tid);
7     pthread_exit(NULL);
8  }
9  int main(int argc, char *argv[])
10 {
11    pthread_t threads[NUM_THREADS];
12    int rc;
13    long t;
14    for(t=0;t<NUM_THREADS;t++)
15    {
16      rc = pthread_create(&threads[t],
17      NULL, PrintHello, (void *)t);
18    }
19    // need to join before exiting...
20 }
```

- Explicitly create and destroy threads.
- Synchronization through mutex, barrier, condition variables.
- Communication via Shared memory – threads can access each others variables.
- Low level interface.
- Supports both MPMD and SPMD.

See `pthread.c` example program

OpenMP

```
1   int i, chunk;
2   float a[N], b[N], c[N];
3
4   /* Some initializations */
5   for (i=0; i < N; i++)
6      a[i] = b[i] = i * 1.0;
7   chunk = CHUNKSIZE;
8
9   #pragma omp parallel shared(a,b,c,chunk) private(i)
10  {
11     #pragma omp for schedule(dynamic,chunk) nowait
12     for (i=0; i < N; i++)
13       c[i] = a[i] + b[i];
14  }   /* end of parallel region */
```

- Higher level interface
- Maps best to shared memory architectures
- Implicit synchronization
- Designed for *data parallel* processing (SPMD).
- Implemented by compiler, designed to be ignored by compilers that don't support it.

See `openmp.c` example program.

MPI

```
1  // Get the number of processes
2  int world_size;
3  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
4  // Get the rank of the process
5  int world_rank;
6  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
7  // Print off a hello world message
8  printf("Hello world from processor %s, rank %d"
9             " out of %d processors\n",
10            processor_name, world_rank, world_size);
11 // Finalize the MPI environment.
12 MPI_Finalize();
```

- Commonly used for large supercomputers
  (distributed memory hw is easier to scale)
- Communication through explicit messages (send/recv) or explicit operations (RMA)
  Distributed memory programming – you can only directly access your own memory
- Usually SPMD.

See `mpi.c` example program.

OpenCL/CUDA

```
1   __global__
2   void saxpy(int n, float a, float *x, float *y)
3   { int i = blockIdx.x*blockDim.x + threadIdx.x;
4     if (i < n) y[i] = a*x[i] + y[i];
5   }
6
7   int main(void)
8   {
9     int N = 1<<20;
10    float *x, *y, *d_x, *d_y;
11    x = (float*)malloc(N*sizeof(float));
12    y = (float*)malloc(N*sizeof(float));
13
14    cudaMalloc(&d_x, N*sizeof(float));
15    cudaMalloc(&d_y, N*sizeof(float));
16    ...
17    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
18    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
19
20    // Perform SAXPY on 1M elements
21    saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
22    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
23    ...
24  }
```

- Similar to OpenMP – compiler assisted.
- Explicit management of accelerator(GPU) and host memory (slowly disappearing).
- Designed for SPMD (data parallel).

See `cuda.c` example program.

Thread Creation/Destruction

Linking with pthreads, creating a thread (`pt1.c`)

Threads

Thread-safe functions (2)

```
1  void * found;
2  bool find_node(node_t * node, void * d)
3  {
4     if (!node)
5        return false;
6     if (node->data==d)
7     {
8        found=node;
9        return true;
10    }
11    else if (find_node(node->left, d))
12    {
13       return true;
14    }
15    else
16      return find_node(node->right, d));
17 }
```

```
1  int pseudorandom()
2  {
3     static_assert(sizeof(int)>=4,
4       "proper size int");
#define M 2147483647
```

For a function to be thread-safe, at the very least, any access to a *shared* resource (memory or other) should be properly synchronized/protected.

Are these examples thread-safe? Why/Why not?

Homework Review

HW6 review