

Midterm II Exam

15-122 Principles of Imperative Computation
Frank Pfenning, Tom Cortina, William Lovas

November 4, 2010

Name: **Sample Solution** Andrew ID: **fp** Section:

Instructions

- This exam is closed-book with one sheet of notes permitted.
- You have 80 minutes to complete the exam.
- There are 4 problems.
- Read each problem carefully before attempting to solve it.
- Consider writing out programs on scratch paper first.

	Ropes	Red/black trees	BDDs	Heapsort	
	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score	40	40	30	30	150
Max	40	40	30	40	150
Grader	fp	tc/wjl	wjl	tc	

1 Ropes (40 pts)

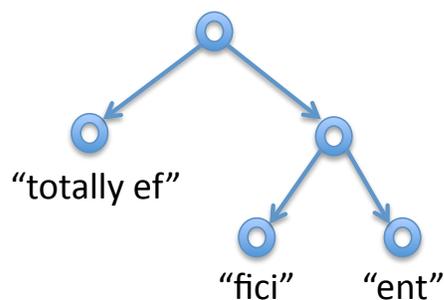
In C0 and C, strings are typically represented as arrays of characters. This allows constant-time access of a character at an arbitrary position, but it also has some disadvantages. In particular, concatenating two strings (function `string_join`) is an expensive operation since we have to create a new character array and copy the two given strings into the new array character by character.

Task 1 (10 pts). What is the asymptotic complexity of the following loop as a function of n ? Assume that `string_fromint` is a constant-time operation.

```
string string_fromarray(int[] A, int n)
//@requires 0 <= n && n < \length(A);
{ string s = "["; int i;
  for (i = 0; i < n; i++)
    //@loop_invariant 0 <= i && i <= n;
    s = string_join(s, string_fromint(A[i]));
  return string_join(s, "]");
}
```

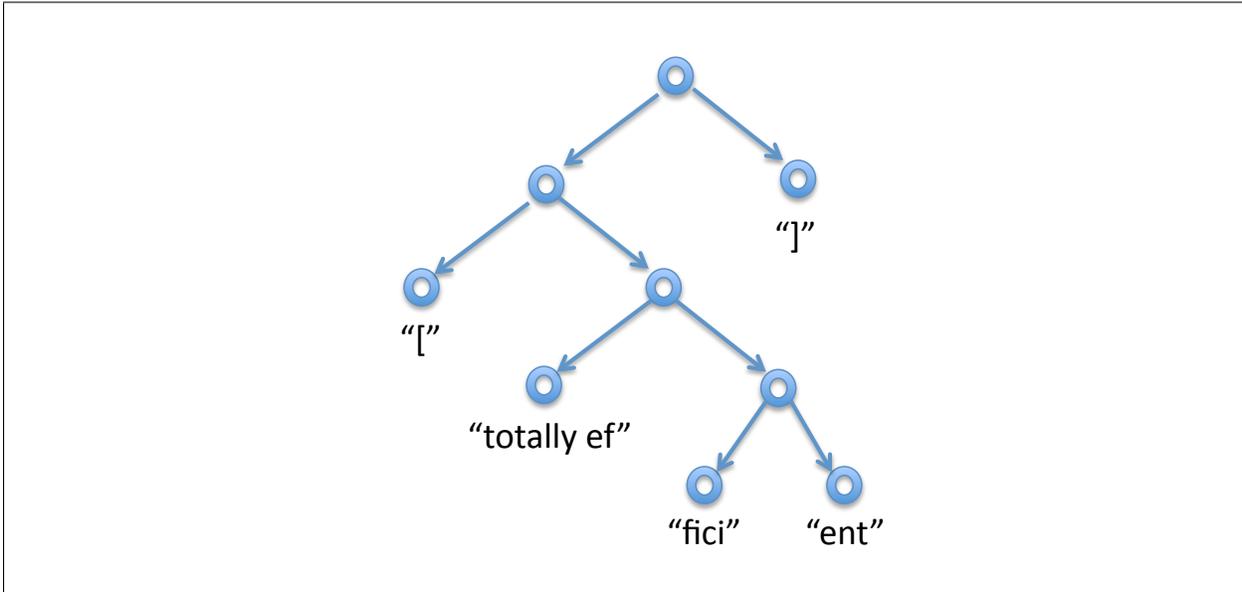
The length of the result of `string_fromint` is bounded by a constant c , and the complexity of `string_join` is $O(k)$, where k is the number of characters in the result. Therefore the complexity is $O(1 * c + 2 * c + 3 * c + \dots + n * c) = O(n^2 * c) = O(n^2)$.

The data structure of *ropes* attempts to improve efficiency of concatenation by representing strings as binary trees, where the leaves contain ordinary strings and the intermediate nodes represent concatenations. For example, the string "totally efficient" might look as follows (among many other possibilities):



Note that ordinary strings are only stored at the leaves. Assuming no rebalancing, concatenation of ropes is a constant-time operation.

Task 2 (10 pts). Assuming no rebalancing, show the final result of concatenating the rope above first on the left with "[" then on the right with "]" to form a rope for the string "[totally efficient]".



Note: Leaf insertion, as we have done for ordinary binary search trees, would require $O(\log(n))$ steps even for balanced trees, rather than $O(1)$ as explained.

Task 3 (10 pts). Describe what additional information you might store in the nodes so that accessing the i th character in a string of length n represented by a rope is $O(\log(n))$ if the rope is balanced.

Store the total length of the string represented by the tree at every node. Looking up an index i goes to the left if the index is strictly less than the length l stored in the left subtree. If $i \geq l$ we go to the right subtree and look up index $i - l$. An index is out of bounds at the root if it is negative or greater or equal to the stored length.

Note: Several variations are possible, but storing the absolute character range for the substring stored in a rope would require an unacceptable $O(n)$ traversal of the right tree after each concatenation to restore this invariant.

Task 4 (10 pts). Carefully describe the invariant for your data structure. You do not need to write code to check it, but your description should be precise and detailed enough that it would be clear how to write the code.

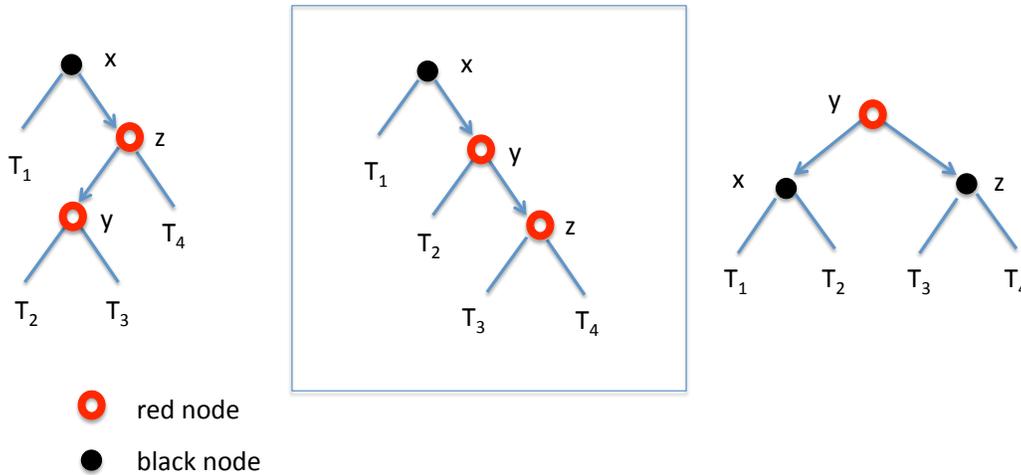
We check that ropes are not empty, that non-empty strings are only stored at leaf nodes (with no children), and that interior nodes have exactly two children.

The length invariant is that each node contains exactly the total length of the string represented by its tree. We can check this at the leaves by comparing it with the length of the string stored there. At interior nodes we check that the stored length is equal to the lengths stored at the subtrees.

2 Red/Black Trees (40 pts)

The transformation below, from the tree on the left to the tree on the right, we called a *double rotation*. It was used in rebalancing after insertion into a red/black tree. We assume keys are integers and x , y , and z are keys for the three nodes shown explicitly.

Task 1 (10 pts). Show that the name *double rotation* is justified by drawing an intermediate tree between the two, so that each step is a single rotation. Try to obey the data structure invariants as much as possible, but see Task 3.



Task 2 (20 pts). Assume that first tree satisfies all red/black tree invariants except for the color invariant at y (whose parent is also red). Also assume that the first tree has no restriction on keys at the root (interval $(-\infty, +\infty)$) and black height h .

Fill in the following table, indicating the bounding intervals and the black height for each subtree.

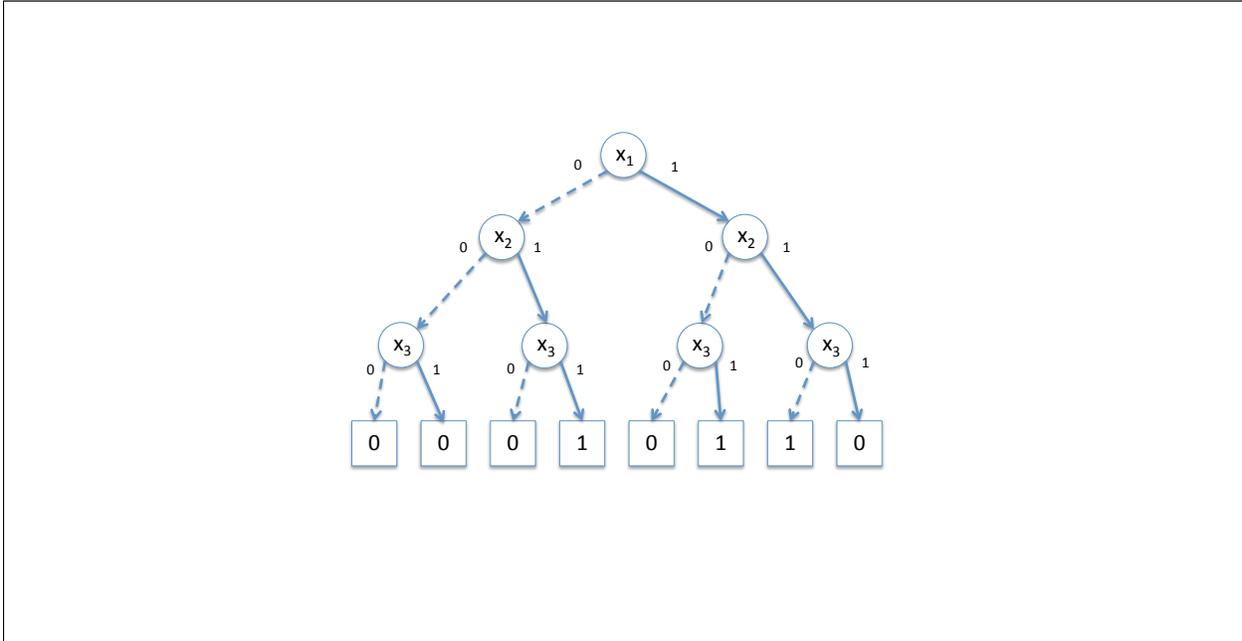
Tree	Interval	Black Height
T_1	$(-\infty, x)$	$h - 1$
T_2	(x, y)	$h - 1$
T_3	(y, z)	$h - 1$
T_4	$(z, +\infty)$	$h - 1$

Task 3 (10 pts). Under the same assumptions as in Task 2, explain which red/black tree invariants hold for the tree you drew in the middle, and which ones are violated and where.

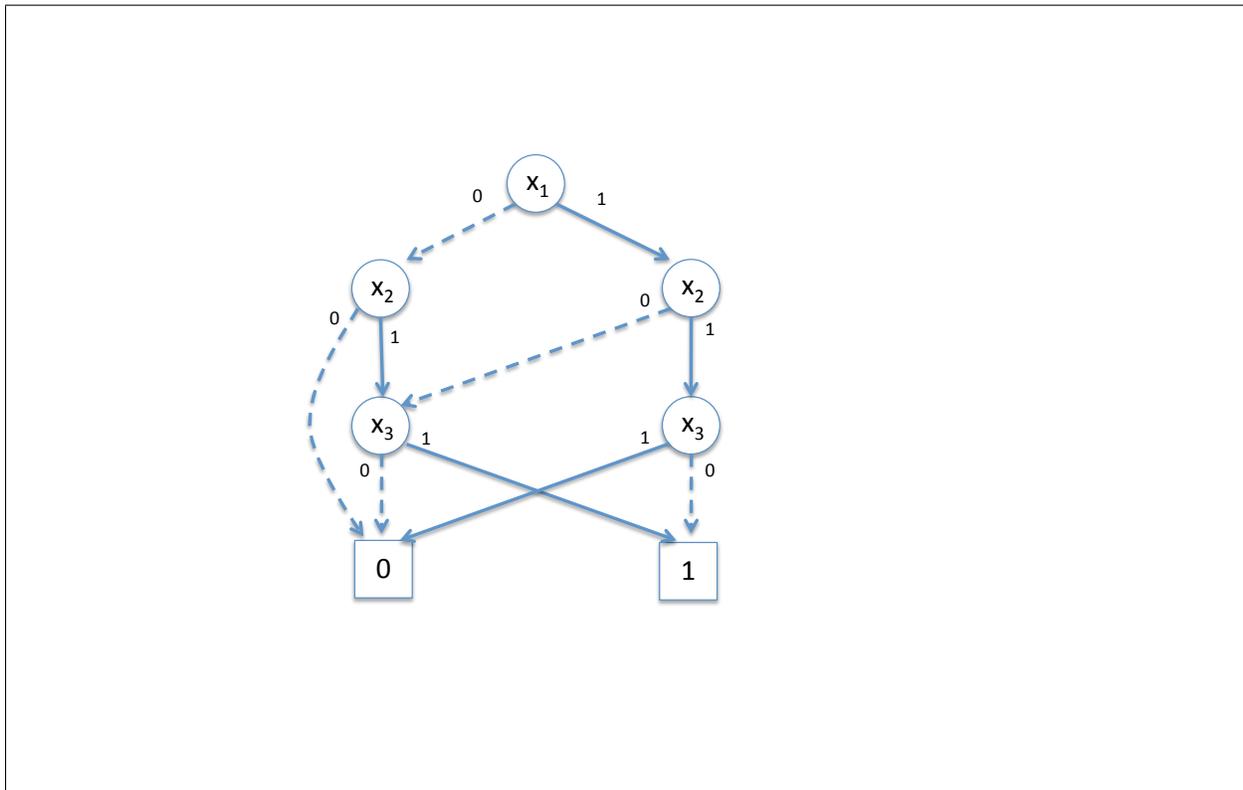
All invariants holds (ordering, black height, color), except for the color invariant at node z which is red and has a red parent y .

3 Binary Decision Diagrams (30 pts)

Task 1 (10 pts). Draw a *binary decision tree* (not a BDD) for the boolean function with arguments x_1 , x_2 , and x_3 that returns true if and only if exactly two of the inputs are true. It should test the variables in the given order.



Task 2 (20 pts). Draw a reduced ordered binary decision diagram (ROBDD) for the boolean function above, again using the given variable ordering. Show intermediate steps in the reduction from the tree to the ROBDD for partial credit in case your final result is not quite correct.



4 Heapsort (40 pts)

We can use the invariant behind heaps in order to implement an in-place sorting algorithm for arrays called *heapsort*. For simplicity, we use *max* heaps, which satisfy:

Max Heap Ordering Invariant: Each node except for the root must be *less or equal* to its parent.

This guarantees that a *maximal* element is at the root of the heap, rather than a minimal one as we did in lecture.

The algorithm proceeds in two phases. In phase one we build up a heap spanning the whole array, in phase two we successively delete the maximum element from the heap and move it the end.

Here is our implementation, written compactly, with only pre- and post-conditions, but no loop invariants or assertions. Note that we only sort the range $A[1, n)$, ignoring $A[0]$.

```
void heapsort(int[] A, int n)
//@requires 1 <= n && n <= \length(A);
//@ensures is_sorted(A, 1, n);
{ int i;
  for (i = 2; i < n; i++) {
    sift_up(A, i, i+1);
  }
  for (i = n-1; 2 <= i; i--) {
    swap(A, 1, i);
    sift_down(A, 1, i);
  }
}
```

The functions `sift_up` and `sift_down` are like the functions we wrote in lecture, except that they take an array as a first argument and what we called $H \rightarrow \text{next}$ (the index right after the last element currently in the heap) as the third argument. The pre- and post-conditions for both functions are given below.

Your main task will be to enrich this code with invariants and assertions. You should assume the following functions:

```
bool is_heap(int[] A, int n);
bool is_heap_except_up(int[] A, int i, int n);
bool is_heap_except_down(int[] A, int i, int n);
bool is_sorted(int[] A, int lower, int upper);
```

with the following interpretation:

`is_heap(A, n)` means that the range $A[1, n)$ satisfies the heap invariant.

`is_heap_except_up(A, i, n)` means that the range $A[1, n)$ satisfies the heap invariant except that $A[i]$ (which must be in the heap) may be greater than its parent.

`is_heap_except_down(A, i, n)` means that the range $A[1, n)$ satisfies the heap invariant except that $A[i]$ (which must be in the heap) may be less than one or both of its children.

`is_sorted(A, lower, upper)` means that the range $A[lower, upper)$ is sorted in increasing order.

Here are the pre- and post-conditions for the `sift_up` and `sift_down` functions that are called from `heapsort`.

```
void sift_up(int[] A, int i, int n)
//@requires is_heap_except_up(A, i, n);
//@ensures is_heap(A, n);
;

void sift_down(int[] A, int i, int n)
//@requires is_heap_except_down(A, i, n);
//@ensures is_heap(A, n);
;
```

Task 1 (25 pts). The following is a correct implementation of `heapsort`, which sorts the range $A[1, n)$ in place. Fill in the strongest correct annotations in the given places, using only the functions `is_heap`, `is_heap_except_up`, `is_heap_except_down` and `is_sorted`. To give you a head start we have included loop index invariants already.

```
void heapsort(int[] A, int n)
//@requires 1 <= n && n <= \length(A);
//@ensures is_sorted(A, 1, n);
{ int i;
  for (i = 2; i < n; i++)
    //@loop_invariant 2 <= i && i <= n;

    //@loop_invariant -----
    {

      //@assert -----
      sift_up(A, i, i+1);
    }

    //@assert -----
  for (i = n-1; 2 <= i; i--)
    //@loop_invariant 1 <= i && i <= n-1;

    //@loop_invariant -----
    {
      swap(A, 1, i);

      //@assert -----
      sift_down(A, 1, i);
    }
}
```

```

void heapsort(int[] A, int n)
//@requires 1 <= n && n <= \length(A);
//@ensures is_sorted(A, 1, n);
{ int i;
  for (i = 2; i < n; i++)
    //@loop_invariant 2 <= i && i <= n;
    //@loop_invariant is_heap(A, i);
    {
      //@assert is_heap_except_up(A, i, i+1);
      sift_up(A, i, i+1);
    }
  //@assert is_heap(A, n);
  for (i = n-1; 2 <= i; i--)
    //@loop_invariant 1 <= i && i <= n-1;
    //@loop_invariant is_heap(A, i+1) && is_sorted(A, i+1, n);
    {
      swap(A, 1, i);
      //@assert is_heap_except_down(A, 1, i) && is_sorted(A, i, n);
      sift_down(A, 1, i);
    }
}

```

Task 2 (15 pts). Analyze the asymptotic complexity of our version of heapsort.

During phase one, each of the $n - 2$ call to sift-up is bounded by $O(\log(n))$, so phase one is bounded by $O(n * \log(n))$. Similarly, each of the $n - 2$ calls to sift-down is bounded by $O(\log(n))$, so phase two is also bounded by $O(n * \log(n))$, leading to a total of $O(2 * n * \log(n)) = O(n * \log(n))$.