Joseph Salazar Homework #5 Murrell ECE 421

import "io"

static {list\_of\_chunks, list\_idx}

let compare(in, end) be

{

let i =0; let bool = 1;

if strlen(in) - strlen(end) /= 0 then

resultis 0;

while true do

{

if byte i of in /= byte i of end then

{

bool := 0;

break;

}

if byte i of in = 0 then

break;

i := i + 1;

}

resultis bool;

}

let strncpy(dst, src, n) be

{

for i = 0 to n - 1 do

{ let c = byte i of src;

byte i of dst := c;

if c = 0 then

return }

byte (n) of dst := 0

}

manifest

{

node\_str =0,

node\_nxt = 1,

sizeof\_node = 2

}

let insert\_chunk(list, node) be

{

switchon list into

{

case nil:

resultis node;

endcase;

default:

list ! node\_nxt := insert\_chunk(list ! node\_nxt, node);

}

resultis list;

}

let get\_size\_of\_string(in) be

{

let size;

size := strlen(in) / 4;

if strlen(in) rem 4 /= 0 then size +:= 1;

if strlen(in) = 4 then size := 2;

resultis size;

}

let printlist(head) be

{

let size;

if head = nil then return;

size := strlen(head ! node\_str) / 4;

if strlen(head ! node\_str) rem 4 /= 0 then size +:= 1;

if strlen(head ! node\_str) = 4 then size := 2;

out("Link = 0x%x, Str = 0x%x, =%s\n", head, head ! node\_str, head ! node\_str);

printlist(head ! node\_nxt);

}

let my\_freevec(node) be //node is a full node

{

let offset = get\_size\_of\_string(node ! node\_str);

node ! node\_nxt := nil;

list\_of\_chunks := insert\_chunk(list\_of\_chunks, node);

}

let remove(head, str) be

{

let tmp; let prev;

tmp := head; prev := nil;

test (tmp /= nil /\ compare(str, tmp ! node\_str) = 1) then //if the head has the string

{

head := tmp ! node\_nxt;

//my\_freevec(tmp ! (node\_str));

my\_freevec(tmp);

resultis head;

}

else

{

while(tmp /= nil /\ compare(tmp ! node\_str, str) = 0) do //find the node with the string

{

prev := tmp;

tmp := tmp ! node\_nxt;

}

if(tmp = nil) then //the string DNE in the list

resultis head;

prev ! node\_nxt := tmp ! node\_nxt; //previous points to two nodes ahead

my\_freevec(tmp); //free the found string

//freevec(tmp); //free the node

resultis head;

}

resultis head;

}

let removechunk(head, str) be

{

let tmp; let prev;

tmp := head; prev := nil;

test (tmp /= nil /\ compare(str, tmp ! node\_str) = 1) then //if the head has the string

{

head := tmp ! node\_nxt;

freevec(tmp);

freevec(tmp ! node\_str);

freevec(tmp ! node\_nxt);

resultis head;

}

else

{

while(tmp /= nil /\ compare(tmp ! node\_str, str) = 0) do //find the node with the string

{

prev := tmp;

tmp := tmp ! node\_nxt;

}

if(tmp = nil) then //the string DNE in the list

resultis head;

prev ! node\_nxt := tmp ! node\_nxt; //previous points to two nodes ahead

freevec(tmp);

freevec(tmp ! node\_str);

freevec(tmp ! node\_nxt); //free the node

resultis head;

}

resultis head;

}

let my\_newvec(str\_in) be

{

let p;

let tmp = list\_of\_chunks;

if tmp = nil then {

p := newvec(sizeof\_node);

p ! node\_str := str\_in;

p ! node\_nxt := nil;

resultis p;

} //list is empty

while tmp /= nil do{

if strlen(str\_in) <= strlen(tmp ! node\_str) then {

out("Recycling Memory Location: 0x%x\n", tmp);

strncpy(tmp ! node\_str, str\_in, strlen(str\_in));

tmp ! node\_nxt := nil;

list\_of\_chunks := removechunk(list\_of\_chunks, tmp ! node\_str);

resultis tmp; // return node

}

tmp := tmp ! node\_nxt;

}

p := newvec(sizeof\_node);

p ! node\_str := str\_in;

p ! node\_nxt := nil;

resultis p;

//here, we add to heap because no recycle available

}

let new\_node(x) be

{

let p = my\_newvec(x);

//p ! node\_str := x;

//p ! node\_nxt := nil;

resultis p;

}

let insert(node, str) be

{

switchon node into

{

case nil:

resultis new\_node(str);

endcase;

default:

node ! node\_nxt := insert(node ! node\_nxt, str);

}

resultis node;

}

let get\_del(s) be

{

let len = strlen(s) - 7; let i = 7; let str;

len := (len / 4) + 1;

str := newvec(len);

while byte i of s /= 0 do

{

byte i-7 of str := byte i of s;

i+:=1;

}

resultis str;

}

let del(comp, del) be

{

let str;

for i = 0 to 5 do

{

if(byte i of comp) /= (byte i of del) then

resultis 0; // not delete

}

resultis 1; //Means delete

}

let all(comp, al) be

{

let str;

for i = 0 to 2 do

{

if(byte i of comp) /= (byte i of al) then

resultis 0; // not all

}

resultis 1; //Means all

}

let mystrdup(s, length) be

{

let newlength; let r;

newlength := length/4 + 1;

r := newvec(newlength);

for i = 0 to newlength - 1 do

r!i := s!i;

//out("%s\n",r);

resultis r;

}

let string\_in() be {

let s = vec(100);

let a = 0; let i = 0;

while i < 399 do {

let char = inch();

if char = '\n' then

break;

byte i of s := char;

i +:= 1; a +:= 1;

byte i of s := 0;

}

resultis mystrdup(s, i);

}

let start() be

{

let str;

let flag;

let links = nil;

let tmp;

let heap = vec(10000);

init(heap, 10000);

list\_of\_chunks := nil;

out("Address of heap is %x\n", @(heap));

out("Enter the strings and deletions: \n");

while true do

{

str := string\_in();

test(del(str,"DELETE") = 1) then

{

links := remove(links, get\_del(str));

}

else test(all(str,"ALL") = 1) then

{

printlist(links);

}

else test(compare(str, "END") = 0) then

{

links := insert(links, str);

}

else

{

break;

}

}

printlist(links);

freevec(links);

}

Text, letter

Description automatically generated