ECE421 Assignment 5

Brandon Rubio C23727769

Files: /home/students/bur2/OS/Assignments/Assignment5/heap.b

/home/students/bur2/OS/Assignments/Assignment5/hw4.b

**TEST RUNS**

Top level test of hw4 with heap.b

Enter the strings and deletions:

hello

this

program

works

ALL

(link@0xFFFC0)(string@0xFFFC0)works

(link@0xFFFD0)(string@0xFFFD0)program

(link@0xFFFE0)(string@0xFFFE0)this

(link@0xFFFF0)(string@0xFFFF0)hello

properly

DELETE

Enter word to delete

program

dog

ALL

(link@0xFFFD0)(string@0xFFFD0)dog

(link@0xFFFB0)(string@0xFFFB0)properly

(link@0xFFFC0)(string@0xFFFC0)works

(link@0xFFFE0)(string@0xFFFE0)this

(link@0xFFFF0)(string@0xFFFF0)hello

cat

DELETE

Enter word to delete

dog

ALL

(link@0xFFFA0)(string@0xFFFA0)cat

(link@0xFFFB0)(string@0xFFFB0)properly

(link@0xFFFC0)(string@0xFFFC0)works

(link@0xFFFE0)(string@0xFFFE0)this

(link@0xFFFF0)(string@0xFFFF0)hello

END

Test of heap.b with code provided in class 13 “Testing a heap allocation system”

Note: actual test code was slightly modified to work with my heap program. The test file that derived these outputs can be found in the same directory as heap.b and hw.b as mentioned above. The file is called “test.b”

x to exit.

?N to examine any chunk. N is its real address, which is the value returned by newvec minus two.

vf to freevec the variable named v. Variable names are single lower-case letters.

v? to see the value of variable v.

vnN to newvec(N), N is number of words wanted, v is the variable in which the returned value will be stored,

the response is v = A, where A is the pointer returned by newvec.

summary of freelist

size: >= 8000, total: 1

(0x1C7D) size = 1041280

> an1041000

a = 7574

7572: B8F47C 1041000 ... 1041000

summary of freelist

size: 280, total: 1

(0x1C7D) size = 280

> an bn1

b = 7566

7564: B8F47C 8 ... 8

summary of freelist

size: 272, total: 1

(0x1C7D) size = 272

> cn9

c = 7550

7548: B8F47C 16 ... 16

summary of freelist

size: 256, total: 1

(0x1C7D) size = 256

> dn32

d = 7518

7516: B8F47C 32 ... 32

summary of freelist

size: 224, total: 1

(0x1C7D) size = 224

> en100

e = 7414

7412: B8F47C 104 ... 104

summary of freelist

size: 120, total: 1

(0x1C7D) size = 120

> fn8

f = 7406

7404: B8F47C 8 ... 8

summary of freelist

size: 112, total: 1

(0x1C7D) size = 112

> gn16

g = 7390

7388: B8F47C 16 ... 16

summary of freelist

size: 96, total: 1

(0x1C7D) size = 96

> hn80

h = 7310

7308: B8F47C 80 ... 80

summary of freelist

size: 16, total: 1

(0x1C7D) size = 16

> j in16

i = 7294

7292: B8F47C 16 ... 16

summary of freelist

> a?

7572: B8F47C, USED, size = 1041000, ..., 1041000

> b?

7564: B8F47C, USED, size = 8, ..., 8

> c?

7548: B8F47C, USED, size = 16, ..., 16

> d?

7516: B8F47C, USED, size = 32, ..., 32

> e?

7412: B8F47C, USED, size = 104, ..., 104

> f?

7404: B8F47C, USED, size = 8, ..., 8

> g?

7388: B8F47C, USED, size = 16, ..., 16

> h?

7308: B8F47C, USED, size = 80, ..., 80

> i?

7292: B8F47C, USED, size = 16, ..., 16

> ?a

[B[A[B

?a b

?c

?1973

1973: 71E0003

That is not a valid code for the beginning of a chunk

> ?7572

7572: B8F47C, USED, size = 1041000, ..., 1041000

> ?7564

7564: B8F47C, USED, size = 8, ..., 8

> ?7548

7548: B8F47C, USED, size = 16, ..., 16

> ?7516

7516: B8F47C, USED, size = 32, ..., 32

> ?7412

7412: B8F47C, USED, size = 104, ..., 104

> ?7404

7404: B8F47C, USED, size = 8, ..., 8

> ?7388

7388: B8F47C, USED, size = 16, ..., 16

> ?7308

7308: B8F47C, USED, size = 80, ..., 80

> ?7292

7292: B8F47C, USED, size = 16, ..., 16

> vfd df

summary of freelist

size: 32, total: 1

(0x1D5D) size = 32

> a bf

summary of freelist

size: 8, total: 1

size: 32, total: 1

(0x1D8D) size = 8

(0x1D5D) size = 32

> ff

summary of freelist

size: 8, total: 2

size: 32, total: 1

(0x1CED) size = 8

(0x1D8D) size = 8

(0x1D5D) size = 32

> gf

summary of freelist

size: 8, total: 1

size: 24, total: 1

size: 32, total: 1

(0x1D8D) size = 8

(0x1CDD) size = 24

(0x1D5D) size = 32

> if

summary of freelist

size: 8, total: 1

size: 16, total: 1

size: 24, total: 1

size: 32, total: 1

(0x1D8D) size = 8

(0x1C7D) size = 16

(0x1CDD) size = 24

(0x1D5D) size = 32

> hf

summary of freelist

size: 8, total: 1

size: 32, total: 1

size: 120, total: 1

(0x1D8D) size = 8

(0x1D5D) size = 32

(0x1C7D) size = 120

> cf

summary of freelist

size: 56, total: 1

size: 120, total: 1

(0x1D5D) size = 56

(0x1C7D) size = 120

> df

That is not a used chunk

> ef

summary of freelist

size: 280, total: 1

(0x1C7D) size = 280

> af

summary of freelist

size: >= 8000, total: 1

(0x1C7D) size = 1041280

> x

**CODE**

“heap.b”

import "io"

/\*

heapSizeMax is normally around 1.05 million,

let max n = 999 (arbitrary)

freelist[n] = 8n + 8

where 8n + 8 is size of chunk stored

freelist[999] = 8000

any chunk larger than 8000 will be stored at freelist[999];

\*/

manifest {

freelistSizeMax = 1000, minChunkSize = 8,

bsize\_i = -1, usedOrFree\_i = 0, next\_i = 1, prev\_i = 2, // for chunk addressing, bsize = size at beginning of chunk

free\_id = 98989898, used\_id = 12121212

}

static {freelist = vec(freelistSizeMax), heap, heapSizeMax = 0, heapChunk}

let is\_in\_heap(addr) be { // if addr is in the heap addressing range

if addr >= heap /\ addr <= heap + heapSizeMax - 1 then {

resultis 1;

}

resultis 0;

}

let is\_in\_range(caddr) be { // if caddr is in valid chunk addressing range

//chunk addresses are only valid between heapChunk and heapChunk heapSizeMax - 2 + (2 - minChunkSize)

if caddr >= heapChunk /\ caddr <= (heapChunk + heapSizeMax - 2 + (2 - minChunkSize)) then { // -2 since heapChunk starts at heap + 1

resultis 1;

}

resultis 0;

}

let test.is\_in\_range() be {

out("heap = %d, heapSizeMax = %d, heap end = %d\n", heap, heapSizeMax, heap + heapSizeMax - 1);

for i = heap - 10 to heap + heapSizemax + 9 do {

if i = heap then {

out("caddr = %d, is\_in\_range = %d <-- heap starts here\n", i, is\_in\_range(i));

loop;

}

if i = heap + 1 then {

out("caddr = %d, is\_in\_range = %d <-- first valid caddr\n", i, is\_in\_range(i));

loop;

}

if i = heap + heapSizeMax - 1 then {

out("caddr = %d, is\_in\_range = %d <-- heap ends here\n", i, is\_in\_range(i));

loop;

}

if i = heap + heapSizeMax - 1 + (2 - minChunkSize) then {

out("caddr = %d, is\_in\_range = %d <-- last valid caddr\n", i, is\_in\_range(i));

loop;

}

if is\_in\_range(i) = 0 then {

out("caddr = %d, is\_in\_range = %d\n", i, is\_in\_range(i));

}

}

}

let is\_free(caddr) be { // if caddr is set to free

test is\_in\_range(caddr) then {

if (caddr ! usedOrFree\_i) = free\_id then {

resultis 1;

}

resultis 0;

}

else {

resultis 0;

}

}

let is\_used(caddr) be { // if caddr is set to used

test is\_in\_range(caddr) then {

if (caddr ! usedOrFree\_i) = used\_id then {

resultis 1;

}

resultis 0;

}

else {

resultis 0;

}

}

let is\_valid(caddr) be { // if caddr is a valid chunk address

if not is\_in\_range(caddr) \/ (not is\_free(caddr) /\ not is\_used(caddr)) then {

resultis 0;

}

resultis 1;

}

let left\_of(caddr) be { // returns caddr of chunk to the left of given chunk, -1 otherwise

test is\_valid(caddr) /\ is\_in\_heap(caddr - 2) then {

if is\_valid(caddr -(caddr ! -2)) then {

resultis (caddr -(caddr ! -2));

}

resultis -1;

}

else {

resultis -1;

}

}

let right\_of(caddr) be { // returns caddr of chunk to the right of given chunk, -1 otherwise

test is\_valid(caddr) /\ is\_in\_heap(caddr + bsize\_i) then {

if is\_valid(caddr + (caddr ! bsize\_i)) then {

resultis (caddr + (caddr ! bsize\_i));

}

resultis -1;

}

else {

resultis -1;

}

}

let is\_left\_free(caddr) be {

let r = left\_of(caddr);

if r = -1 then {

resultis 0;

}

if is\_free(r) then {

resultis 1;

}

resultis 0;

}

let is\_right\_free(caddr) be {

let r = right\_of(caddr);

if r = -1 then {

resultis 0;

}

if is\_free(r) then {

resultis 1;

}

resultis 0;

}

let are\_adjacent(caddr1, caddr2) be {

if left\_of(caddr1) = caddr2 \/ right\_of(caddr1) = caddr2 then {

resultis 1;

}

resultis 0;

}

let clear\_chunk\_data(caddr) be {

if is\_free(caddr) then {

for i = 3 to (caddr ! bsize\_i) - 3 do {

caddr ! i := nil;

}

}

if is\_used(caddr) then {

for i = 1 to (caddr ! bsize\_i) - 3 do {

caddr ! i := nil;

}

}

}

let write\_to\_used\_chunk(caddr, amount, data) be { //only for testing

// relative to usesdOrFree\_i = 0

let usedDataStart\_i = 1;

if not is\_valid(caddr) then {

out("invalid caddr (write\_to\_used\_chunk)\n");

return;

}

if is\_free(caddr) then {

out("only use on used chunks (write\_to\_used\_chunk)\n");

return;

}

for i = usedDataStart\_i to amount + usedDataStart\_i - 1 do {

if i = caddr + (caddr ! bsize\_i) - 2 then {

break;

}

caddr ! i := data;

}

}

let show\_chunk\_data(caddr) be {

// relative to usesdOrFree\_i = 0

let usedDataStart\_i = 1;

let freeDataStart\_i = 3;

let dataEnd\_i = (caddr ! bsize\_i) - 3;

if not is\_valid(caddr) then {

out("invalid caddr (show\_chunk\_data)\n");

return;

}

if is\_free(caddr) then {

for i = freeDataStart\_i to dataEnd\_i do {

if i = freeDataStart\_i then {

out("(0x%x) 0x%x 0d%d [%s] <- data start", caddr + i, caddr ! i, caddr ! i, caddr ! i);

test caddr ! i = nil then {

out(" <- nil\n");

}

else {

out("\n");

}

loop;

}

if i = dataEnd\_i then {

out("(0x%x) 0x%x 0d%d [%s] <- data end", caddr + i, caddr ! i, caddr ! i, caddr ! i);

test caddr ! i = nil then {

out(" <- nil\n");

}

else {

out("\n");

}

loop;

}

out("(0x%x) 0x%x 0d%d [%s]", caddr + i, caddr ! i, caddr ! i, caddr ! i);

test caddr ! i = nil then {

out(" <- nil\n");

}

else {

out("\n");

}

}

}

if is\_used(caddr) then {

for i = usedDataStart\_i to dataEnd\_i do {

if i = usedDataStart\_i then {

out("(0x%x) 0x%x 0d%d [%s]<- data start", caddr + i, caddr ! i , caddr ! i, caddr ! i);

test caddr ! i = nil then {

out(" <- nil\n");

}

else {

out("\n");

}

loop;

}

if i = dataEnd\_i then {

out("(0x%x) 0x%x 0d%d [%s]<- data end", caddr + i, caddr ! i, caddr ! i, caddr ! i);

test caddr ! i = nil then {

out(" <- nil\n");

}

else {

out("\n");

}

loop;

}

out("(0x%x) 0x%x 0d%d [%s]", caddr + i, caddr ! i, caddr ! i, caddr ! i);

test caddr ! i = nil then {

out(" <- nil\n");

}

else {

out("\n");

}

}

}

}

let init\_chunk(caddr, usedOrFree, next, prev, size) be { // caddr = chunk address, usedOrFree: used = 1 free = 0,

// relative to usesdOrFree\_i = 0

let usedDataStart\_i = 1;

let freeDataStart\_i = 3;

let dataEnd\_i = size - 3;

let endOfHeap\_i = heapChunk + heapSizeMax - 2;

let endSize\_i = size - 2;

if not is\_in\_range(caddr) then {

out("invalid caddr (init\_chunk)\n");

return;

}

if caddr + size - 2 > endOfHeap\_i then {

out("chunk is too large (init\_chunk)\n");

return;

}

caddr ! bsize\_i := size;

test usedOrFree then {

caddr ! usedOrFree\_i := used\_id;

for i = usedDataStart\_i to dataEnd\_i do {

caddr ! i := nil;

}

}

else {

caddr ! usedOrFree\_i := free\_id;

caddr ! next\_i := next;

caddr ! prev\_i := prev;

for i = freeDataStart\_i to dataEnd\_i do {

caddr ! i := nil;

}

}

caddr ! endSize\_i := size;

}

let test.clear\_chunk\_data() be {

out("used\n");

init\_chunk(heapChunk, 1, nil, nil, 8);

out("data\n");

show\_chunk\_data(heapChunk);

out("written\n");

write\_to\_used\_chunk(heapChunk, 5, 1234);

out("data\n");

show\_chunk\_data(heapChunk);

out("cleared\n");

clear\_chunk\_data(heapChunk);

show\_chunk\_data(heapChunk);

}

let show\_chunk(caddr) be {

if not is\_valid(caddr) then {

out("invalid caddr (show\_chunk)\n");

return;

}

out("(0x%x) 0d%d <- bsize\n", caddr + bsize\_i, caddr ! bsize\_i);

out("(0x%x) 0d%d <- usedOrFree\n", caddr + usedOrFree\_i, caddr ! usedOrFree\_i);

if (caddr ! usedOrFree\_i = free\_id) then {

out("(0x%x) 0d%d <- next\n", caddr + next\_i, caddr ! next\_i);

out("(0x%x) 0d%d <- prev\n", caddr + prev\_i, caddr ! prev\_i);

show\_chunk\_data(caddr);

}

if (caddr ! usedOrFree\_i = used\_id) then {

show\_chunk\_data(caddr);

}

out("(0x%x) 0d%d <- end size\n", caddr + ((caddr ! bsize\_i) - 2), caddr ! ((caddr ! bsize\_i) - 2));

}

let test.init\_chunk() be {

out("free\n");

init\_chunk(heapChunk, 0, 1, 2, heapSizeMax);

show\_chunk(heapChunk);

out("used\n");

init\_chunk(heapChunk, 1, nil, nil, heapSizeMax);

show\_chunk(heapChunk);

}

let test.is\_free\_and\_is\_used() be {

if is\_free(heapChunk) then {

out("heap is free\n");

}

init\_chunk(heapChunk, 1, nil, nil, heapSizeMax);

if is\_used(heapChunk) then {

out("heap is used\n");

}

}

let change\_to\_free(caddr) be { // must have been initialized previously

init\_chunk(caddr, 0, nil, nil, caddr ! bsize\_i);

}

let change\_to\_used(caddr) be { // must have been initialized previously

init\_chunk(caddr, 1, nil, nil, caddr ! bsize\_i);

}

let init\_freelist() be {

for i = 0 to freelistSizeMax - 1 do {

freelist ! i := nil;

}

}

let add\_to\_freelist(caddr) be {

let n, top;

if not is\_valid(caddr) then {

out("invalid caddr (add\_to\_freelist)\n");

return;

}

if not is\_free(caddr) then {

out("caddr is not free (add\_to\_freelist)\n");

return;

}

if ((caddr ! bsize\_i) rem minChunkSize) <> 0 then {

out("size of chunk at caddr = %d is not a multiple of 8 (add\_to\_freelist)\n", caddr);

return;

}

//add to the free list

test (caddr ! bsize\_i) >= minChunkSize /\ (caddr ! bsize\_i) < (minChunkSize \* freelistSizeMax) then {

n := ((caddr ! bsize\_i) - minChunkSize) / minChunkSize;

top := freelist ! n;

test top = nil then {

freelist ! n := caddr;

caddr ! next\_i := nil;

caddr ! prev\_i := nil;

}

else {

top ! prev\_i := caddr;

caddr ! next\_i := top;

caddr ! prev\_i := nil;

freelist ! n := caddr;

}

}

else test (caddr ! bsize\_i) >= (minChunkSize \* freelistSizeMax) then {

n := freelistSizeMax - 1;

top := freelist ! n;

test top = nil then {

freelist ! n := caddr;

caddr ! next\_i := nil;

caddr ! prev\_i := nil;

}

else {

top ! prev\_i := caddr;

caddr ! next\_i := top;

caddr ! prev\_i := nil;

freelist ! n := caddr;

}

}

else {

out("invalid caddr size (add\_to\_freelist)\n");

}

}

let remove\_from\_freelist(caddr) be {

let n, top;

if not is\_in\_range(caddr) then {

out("caddr not in range (remove\_from\_freelist)\n");

return;

}

if ((caddr ! bsize\_i) rem minChunkSize) <> 0 then {

out("size of chunk at caddr = %d is not a multiple of 8 (remove\_from\_freelist)\n", caddr);

return;

}

test (caddr ! bsize\_i) > 0 /\ (caddr ! bsize\_i) < (minChunkSize \* freelistSizeMax) then {

n := ((caddr ! bsize\_i) - minChunkSize) / minChunkSize;

top := freelist ! n;

test top = caddr /\ caddr ! next\_i = nil then {

freelist ! n := nil;

}

else test top = caddr then {

(caddr ! next\_i) ! prev\_i := caddr ! prev\_i;

freelist ! n := caddr ! next\_i;

}

else test caddr ! next\_i = nil then {

(caddr ! prev\_i) ! next\_i := nil;

}

else {

(caddr ! prev\_i) ! next\_i := caddr ! next\_i;

(caddr ! next\_i) ! prev\_i := caddr ! prev\_i;

}

caddr ! prev\_i := nil;

caddr ! next\_i := nil;

}

else test caddr ! bsize\_i >= (minChunkSize \* freelistSizeMax) then {

n := freelistSizeMax - 1;

top := freelist ! n;

test top = caddr /\ caddr ! next\_i = nil then {

freelist ! n := nil;

}

else test top = caddr then {

(caddr ! next\_i) ! prev\_i := caddr ! prev\_i;

freelist ! n := caddr ! next\_i;

}

else test caddr ! next\_i = nil then {

(caddr ! prev\_i) ! next\_i := nil;

}

else {

(caddr ! prev\_i) ! next\_i := caddr ! next\_i;

(caddr ! next\_i) ! prev\_i := caddr ! prev\_i;

}

caddr ! prev\_i := nil;

caddr ! next\_i := nil;

}

else {

out("invalid caddr size (remove\_from\_freelist)\n");

}

}

let test.remove\_from\_freelist() be {

remove\_from\_freelist(heapChunk);

}

let merge\_freeChunks(caddr1, caddr2) be { //returns caddr of new combined free chunk, -1 otherwise

// this shouldn't worry about adding or removing from free list, it only merges two free chunks and checks if they're free and adjacent

// if they're in or outside the free list it does not care

if not is\_valid(caddr1) then {

out("invalid caddr1 (merge\_freeChunks)\n");

resultis -1;

}

if not is\_valid(caddr2) then {

out("invalid caddr2 (merge\_freeChunks)\n");

resultis -1;

}

if not is\_free(caddr1) then {

out("caddr1 is not free (merge\_freeChunks)\n");

resultis -1;

}

if not is\_free(caddr2) then {

out("caddr2 is not free (merge\_freeChunks)\n");

resultis -1;

}

if are\_adjacent(caddr1, caddr2) then {

if left\_of(caddr1) = caddr2 then {

init\_chunk(caddr2, 0, nil, nil, (caddr1 ! bsize\_i) + (caddr2 ! bsize\_i));

clear\_chunk\_data(caddr2);

resultis caddr2;

}

if right\_of(caddr1) = caddr2 then {

init\_chunk(caddr1, 0, nil, nil, (caddr1 ! bsize\_i) + (caddr2 ! bsize\_i));

clear\_chunk\_data(caddr1);

resultis caddr1;

}

}

resultis -1;

}

let merge\_check(caddr) be {

let left, mid, right, r;

if not is\_valid(caddr) then {

out("invalid caddr (merge\_check)\n");

return;

}

if not is\_free(caddr) then {

out("caddr is not free (merge\_check)\n");

return;

}

mid := caddr;

test is\_right\_free(caddr) /\ is\_left\_free(caddr) then {

left := left\_of(mid);

right := right\_of(mid);

if right = -1 then {

out("right = -1 if right /\ left (merge\_check)\n");

return;

}

if left = -1 then {

out("left = -1 if righht /\ left(merge\_check)\n");

return;

}

//remove left, mid, and right from free list

remove\_from\_freelist(left);

remove\_from\_freelist(mid);

remove\_from\_freelist(right);

//merge mid and right

r := merge\_freeChunks(mid, right);

if r = -1 then {

out("r = -1 mid + right right /\ left\n");

return;

}

//merge left and (mid + right)

r := merge\_freeChunks(left, r);

if r = -1 then {

out("r = -1 left + mid&right if right /\ left\n");

return;

}

//add back to free list and check if newly combined chunk can be combined again

add\_to\_freelist(r);

merge\_check(r);

}

else test is\_right\_free(caddr) then {

right := right\_of(mid);

if right = -1 then {

out("right = -1 if right (merge\_check)\n");

return;

}

//remove mid and right from free list

remove\_from\_freelist(mid);

remove\_from\_freelist(right);

//merge mid and right

r := merge\_freeChunks(mid, right);

if r = -1 then {

out("r = -1 mid + right if right (merge\_check)\n");

return;

}

//add back to free list and check if newly combined chunk can be combined again

add\_to\_freelist(r);

merge\_check(r);

}

else test is\_left\_free(caddr) then {

left := left\_of(mid);

if left = -1 then {

out("left = -1 if left (merge\_check)\n");

return;

}

//remove left and mid from free list

remove\_from\_freelist(left);

remove\_from\_freelist(mid);

//merge left and mid

r := merge\_freeChunks(left, mid);

if r = -1 then {

out("r = -1 left + mid if left (merge\_check)\n");

return;

}

add\_to\_freelist(r);

merge\_check(r);

}

else {

return;

}

}

let add\_and\_merge(caddr) be {

add\_to\_freelist(caddr);

merge\_check(caddr);

}

let init\_heap\_chunk() be {

heapChunk := heap + 1;

init\_chunk(heapChunk, 0, nil, nil, heapSizeMax);

add\_to\_freelist(heapChunk);

}

let my\_init() be {

heap := !0x101;

heapSizeMax := !0x100 - !0x101;

// this is so the heap will always be a multiple of 8, allows for no fragments < size 8 when splitting chunks

if heapSizeMax < minChunkSize then {

out("insufficient heapsize\n");

return;

}

if (heapSizeMax rem minChunkSize) <> 0 then {

heapSizeMax -:= heapSizeMax rem minChunkSize;

}

init\_freelist();

init\_heap\_chunk();

}

/\* otherwise

let my\_init(ptr, size) be {

heap := ptr;

heapSizeMax := size;

...

}

\*/

let return\_heap() be {

resultis heap;

}

let return\_heapSizeMax() be {

resultis heapSizeMax;

}

let compatible\_chunkSize(size) be { //returns -1 on error, otherwise a compatible size

//adjusts given size to be > 0 and a multiple of minChunkSize

test size <= 0 then {

out("invalid size (compatible\_chunkSize)\n");

resultis -1;

}

else test not (size rem minChunkSize) then {

resultis size;

}

else {

resultis size + minChunkSize - (size rem minChunkSize);

}

}

let split\_free\_chunk(caddr, size) be { //will split a free chunk and return a free chunk of requested size

let compSize = compatible\_chunkSize(size);

let originalChunk = caddr;

let newChunk;

//is there enough space to split the chunk?

test compSize = -1 \/ ((caddr ! bsize\_i) - compSize) < minChunkSize then {

out("unable to split chunk\n");

resultis -1;

}

else {

remove\_from\_freelist(originalChunk); // remove original chunk from free list

init\_chunk(originalChunk, 0, nil, nil, ((caddr ! bsize\_i) - compSize)); // shrink original chunk

newChunk := originalChunk + (originalChunk ! bsize\_i); // define newChunk

init\_chunk(newChunk, 1, nil, nil, compSize); // initialize new chunk from original (as used so it doesnt get merged)

add\_and\_merge(originalChunk);

init\_chunk(newChunk, 0, nil, nil, size); // re-initialize new chunk as free

add\_to\_freelist(newChunk); // add new chunk back into the list

resultis newChunk; //return newChunk

}

}

let search\_freelist(size) be { //returns caddr of free chunk of selected size from freelist, -1 otherwise

let compSize = compatible\_chunkSize(size), n, ptr;

if compSize = -1 then {

resultis -1;

}

n := (compSize - minChunkSize) / minChunkSize;

if n > freelistSizeMax - 1 then {

n := freelistSizeMax - 1;

}

ptr := freelist ! n;

while ptr <> nil do {

if ptr ! bsize\_i = compSize then { // a double check

resultis ptr;

}

ptr := ptr ! next\_i;

}

resultis -1;

}

let next\_biggest\_free\_chunk(size) be { //returns caddr of next biggest free chunk in freelist, -1 otherwise

let compSize = compatible\_chunkSize(size), n, ptr;

if compSize = -1 then {

resultis -1;

}

n := (compSize - minChunkSize) / minChunkSize;

if n > freelistSizeMax - 1 then {

n := freelistSizeMax - 2; // -2 so that freelist[freelistSizeMax - 1] can be searched in the following for loop

}

for i = (n + 1) to freelistSizeMax - 1 do {

test freelist ! i = nil then {

loop;

}

else {

ptr := freelist ! i;

while ptr <> nil do {

if ptr ! bsize\_i > compSize then {

resultis ptr;

}

ptr := ptr ! next\_i;

}

}

}

resultis -1;

}

let my\_newvec(size) be { //at the moment any given chunk from my\_newvec can be overwritten, end of newvecs must me checked using virtual memory

let compSize = compatible\_chunkSize(size), n, caddr;

if compSize = -1 then {

return; // error message handled by compatible\_chunkSize

}

caddr := search\_freelist(compSize);

test caddr = -1 then { // if search\_freelist fails

caddr := next\_biggest\_free\_chunk(compSize);

test caddr = -1 then { // if no bigger chunk to be split is found

out("no space for newvec (my\_newvec)\n");

return;

}

else { // bigger chunk to be split is found

caddr := split\_free\_chunk(caddr, compSize);

test caddr = -1 then { // split\_free\_chunk error

return; // error message handled by split\_free\_chunk

}

else { // new free chunk is made and returned

remove\_from\_freelist(caddr);

change\_to\_used(caddr);

resultis caddr + 1; // + 1 to point to chunk data

}

}

}

else { // if free chunk of compSize is found

remove\_from\_freelist(caddr);

change\_to\_used(caddr);

resultis caddr + 1; // + 1 to point to chunk data

}

}

let my\_freevec(daddr) be { // daddr = beginning of data section in a chunk

if not is\_valid(daddr - 1) then {

out("invalid daddr (my\_freevec)\n");

return;

}

if not is\_used(daddr -1) then {

out("daddr is not used (my\_freevec)\n");

return;

}

clear\_chunk\_data(daddr - 1);

init\_chunk(daddr - 1, 0, nil, nil, (daddr - 1) ! bsize\_i);

add\_and\_merge(daddr - 1);

}

let total\_free\_chunks\_of\_size(size) be {

let n, count = 0, ptr;

if not (size rem minChunkSize = 0 /\ size > 0) then {

out("invalid chunk size (total\_chunks\_of\_size)\n");

return;

}

n := (size - minChunkSize) / minChunkSize;

ptr := freelist ! n;

while ptr <> nil do {

count +:= 1;

ptr := ptr ! next\_i;

}

resultis count;

}

let freelist\_summary() be {

let size, total;

out("summary of freelist\n");

for n = 0 to freelistSizeMax - 1 do {

size := minChunkSize \* n + minChunkSize;

total := total\_free\_chunks\_of\_size(size);

if total > 0 then {

test size = freelistSizeMax \* minChunkSize then {

out("size: >= %d, total: %d\n", size, total\_free\_chunks\_of\_size(size));

}

else {

out("size: %d, total: %d\n", size, total\_free\_chunks\_of\_size(size));

}

}

}

}

let print\_all\_free\_chunks() be {

let ptr;

for n = 0 to freelistSizeMax - 1 do {

ptr := freelist ! n;

while ptr <> nil do {

out("(0x%x) size = %d\n", ptr, ptr ! bsize\_i);

ptr := ptr ! next\_i;

}

}

}

let printfrees() be {

freelist\_summary();

print\_all\_free\_chunks();

}

let pre\_start() be {

init := my\_init;

newvec := my\_newvec;

freevec := my\_freevec;

init();

}

export {my\_init, my\_freevec, my\_newvec, return\_heap, return\_heapSizeMax, show\_chunk, printfrees}

“hw4.b”

import "io"

import "heap"

manifest {linkSize = 2, bufferSize = 100, data\_i = 0, next\_i = 1} // i = index

let strdup(string) be {

// string must have terminating null character

let stringSize = strlen(string);

let stringSizeWithNull = stringSize + 1;

let cpy;

test not((stringSizeWithNull rem 4)) then {

cpy := newvec(stringSizeWithNull / 4 );

}

else {

cpy := newvec((stringSizeWithNull / 4 ) + 1);

}

//out("strdup string = %s, size = %d\n", string, stringSize);

for i = 0 to stringSize do {

byte i of cpy := byte i of string;

}

resultis cpy;

}

let strcmp(a, b) be {

let i = 0;

//out("entered strcmp, a = %s, b = %s\n", a, b);

while 1 do {

if byte i of a <> byte i of b then {

resultis false;

}

if byte i of a = 0 \/ byte i of b = 0 then {

break;

}

i +:= 1;

}

if byte i of a = 0 /\ byte i of b = 0 then {

resultis true;

}

resultis false;

}

let strncmp(a, b, max) be {

for i = 0 to max do

{ if byte i of a <> byte i of b then

resultis false;

if byte i of a = 0 then

resultis true }

resultis true

}

let strcpy(a, b) be {

// copy from a to b;

let i = 0;

while 1 do {

byte i of b := byte i of a;

if byte i of a = 0 then {

break;

}

i +:= 1;

}

}

let clearBuffer(buf) be {

for i = 0 to (bufferSize \* 4) - 1 do {

byte i of buf := '\0';

}

}

let readline(buffer) be {

//inch() causes isues when string typed > bufferSize

//if statement cuts off input to the buffer when bufferSize is reached

//but inch() will continue to stream its recorded input until its end is reached

//this gets read as a regular terminal input and that last remaining chunk of characters gets

//interpreted as another entry to be added to the linked list

let character, temp;

//clearBuffer(buffer);

for i = 0 to bufferSize do {

character := inch();

//out("buffer length = %d\n", strlen(buffer));

if character = '\n' \/ i = bufferSize then {

byte i of buffer := '\0';

break;

}

byte i of buffer := character;

}

}

let printString(string) be {

out("%s\n", string);

}

let newLink(data, next) be {

//data is a pointer to a vector of characters (string), next is a pointer to another link

//data is the address of the string, must input an address for this to work

let r = newvec(linkSize);

r ! data\_i := data;

r ! next\_i := next;

resultis r;

}

let add(string, topOfList) be {

let cpy = strdup(string);

let newTop;

//out("add(%s, %d)\n", cpy, topOfList);

newTop := newLink(cpy, topOfList);

resultis newTop;

}

let remove(string, topOfList) be { //returns top of list

let temp;

let top = topOfList;

let ptr = topOfList;

test strcmp(string, top ! data\_i) then {

temp := top ! next\_i;

freevec(top ! data\_i);

freevec(top);

resultis temp;

}

else {

while ((ptr ! next\_i) <> nil) do {

test strcmp(string, (ptr ! next\_i) ! data\_i) then {

temp := ptr ! next\_i;

ptr ! next\_i := (ptr ! next\_i) ! next\_i;

freevec(temp ! data\_i);

freevec(temp);

break;

}

else {

ptr := ptr ! next\_i;

}

}

resultis top;

}

resultis top;

}

let allListDetails(topOfList) be {

let ptr = topOfList;

while (ptr <> nil) do {

out("Link\n");

out("(0x%x) 0x%x 0d%d [%s]\n", ptr, ptr ! data\_i, ptr ! data\_i, ptr ! data\_i);

out("Link Chunk Data\n");

show\_chunk(ptr - 1);

out("Link -> next\n");

out("(0x%x) 0x%x 0d%d [%s]\n", ptr + next\_i, ptr ! next\_i, ptr ! next\_i, ptr ! next\_i);

out("Link -> data\n");

out("(0x%x) 0x%x 0d%d [%s]\n", ptr + data\_i, ptr ! data\_i, ptr ! data\_i, ptr ! data\_i);

out("Link -> data Chunk Data\n");

show\_chunk((ptr ! data\_i) - 1);

out("\n");

ptr := ptr ! next\_i;

}

out("\n");

}

let all(topOfList) be {

let ptr = topOfList;

//printfrees();

while (ptr <> nil) do {

out("(link@0x%x)(string@0x%x)%s\n", ptr, ptr + data\_i, ptr ! data\_i);

ptr := ptr ! next\_i;

}

}

let start() be {

let top = nil, i = 1, buffer = vec((bufferSize / 4) + 1);

let result;

let heap = return\_heap;

let heapSizeMax = return\_heapSizeMax;

//out("heap start = 0d%d 0x%x, heapChunk start = 0d%d 0x%x, heap end = 0d%d 0x%x, heapSizeMax = 0d%d 0x%x\n", heap, heap, heap + 1, heap + 1, heap + heapSizeMax - 1, heap + heapSizeMax - 1,heapSizeMax,heapSizeMax);

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

while 1 do {

//out("before readline buffer size: %d\n", strlen(buffer));

readline(buffer);

//out("after readline buffer size: %d\n", strlen(buffer));

//out("buffer = %s\n", buffer);

if strcmp(buffer, "ALL") then {

//out("if all\n");

all(top);

loop;

}

if strcmp(buffer, "DELETE") then {

out("Enter word to delete\n");

readline(buffer);

top := remove(buffer, top);

loop;

}

if strcmp(buffer, "END") then {

//out("break\n");

break;

loop;

}

if strlen(buffer) <> 0 then {

//out("added\n");

top := add(buffer, top);

//out("result: %d, data : %s, next: %d\n", result, result ! data\_i, result ! next\_i);

}

}

}